AFRL-IF-RS-TR-2001-68
Final Technical Report
April 2001

AN OPEN IMPLEMENTATION TOOLKIT FOR CREATING ADAPTABLE DISTRIBUTED APPLICATIONS

BBN Technologies

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. F164

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2001-68 has been reviewed and is approved for publication.

APPROVED: Patrick M. Hurley

PATRICK M. HURLEY
Project Engineer

FOR THE DIRECTOR:

WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

AN OPEN IMPLEMENTATION TOOLKIT FOR CREATING ADAPTABLE DISTRIBUTED APPLICATIONS

Joseph P. Loyall

Contractor:	BBN Technologies
Contract Number:	F30602-97-C-0276
Effective Date of Contract:	15 July 1997
Contract Expiration Date:	14 July 2000
Short Title of Work:	An Open Implementation Toolkit For Creating Adaptable Distributed Applications
Period of Work Covered:	Jul 97 - Jul 00
Principal Investigator:	Joseph P. Loyall
Phone:	(617) 873-4679
AFRL Project Engineer:	Patrick M. Hurley
Phone:	(315) 330-3624

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Patrick M. Hurley, AFRL/IFGA, 525 Brooks Road, Rome, NY.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** APRIL 2001	**3. REPORT TYPE AND DATES COVERED** Final Jul 97 - Jul 00

4. TITLE AND SUBTITLE
AN OPEN IMPLEMENTATION TOOLKIT FOR CREATING ADAPTABLE DISTRIBUTED APPLICATIONS

5. FUNDING NUMBERS
C - F30602-97-C-0276
PE - 62301E
PR - F164
TA - 40
WU - 13

6. AUTHOR(S)
Joseph P. Loyall

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
BBN Technologies
10 Moulton Street
Cambridge MA 02138

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Project Agencies
3701 North Fairfax Drive
Arlington VA 22203-1714

Air Force Research Laboratory/IFGA
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2001-68

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer: Patrick M. Hurley/IFGA/(315) 330-3624

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Open Implementation Toolkit for Creating Adaptable Distributed Applications project developed enabling technologies for building intrusion-aware, adaptable applications, i.e., applications that can recognize intrusions, attacks, and malfunctions, and adapt to avoid them or reduce their impact, thereby increasing their chances for survival. The project developed and packaged these technologies into the QuO Open Implementation Toolkit, containing specification languages and code generators, a runtime kernel, and libraries of reusable components that support measurement, control and adaptation at many levels of system operation. The project also demonstrated these technologies by developing adaptive, survivable example applications using a variety of intrusion detection, security, and property managers. These applications developed using the QuO toolkit, demonstrate the concepts underlying advances in infrastructure to detect potential intrusions and adapt to recover from, avoid, or protect against them.

14. SUBJECT TERMS
Quality of Service, Adaptable, Survivable, Distributed System, Intrusion-aware, Object-Oriented

15. NUMBER OF PAGES
64

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

Table of Figures

List of Tables

Executive Summary

The Open Implementation Toolkit for Creating Adaptable Distributed Applications project developed enabling technologies for building *intrusion-aware, adaptable* applications, i.e., applications that can recognize intrusions, attacks, and malfunctions, and adapt to avoid them or reduce their impact, thereby increasing their chances for survival. The project developed and packaged these technologies into the QuO Open Implementation Toolkit, containing specification languages and code generators, a runtime kernel, and libraries of reusable components that support measurement, control, and adaptation at many levels of system operation. The project also demonstrated these technologies by developing adaptive, survivable example applications using a variety of intrusion detection, security, and property managers. These applications, developed using the QuO toolkit, demonstrate the concepts underlying advances in infrastructure to detect potential intrusions and adapt to recover from, avoid, or protect against them.
The *QuO Open Implementation Toolkit* consists of the following components:

- Description languages for specifying operating ranges, implementation alternatives, adaptation strategies, and the system resources and conditions that must be monitored at runtime to detect possible intrusions, malfunctions, and attacks; and code generators for producing adaptation components from the descriptions. The toolkit provides three description languages: the *contract* description language, the *structure* description language, and the *connection* specification language.
- A runtime kernel to monitor the appropriate system resources and conditions, recognize when an implementation is operating outside its acceptable range, and help the application reconfigure (e.g., change to an alternate implementation) to avoid the problem.
- System condition objects that interface to property managers, mechanisms, and resources, such as intrusion detection systems and dependability managers.
- An ORB gateway shell, for supporting adaptation and control at the inter-ORB transport layer
- Instrumentation support, including support for gathering information, inserting probes, and passing data along with round-trip method calls.

As a major component of this project, we demonstrated the use of the Open Implementation Toolkit for developing adaptive, survivable applications. We developed several example applications that exhibit the following characteristics improving their survivability:

- *Adaptation for survival.* These applications can adapt to changing conditions in their environment, including reported intrusions and changes in security policies. This enables them to avoid potential intrusions, continue in the face of degraded service, and recover from intrusions and faults.
- *Intrusion- and security-awareness.* These applications can aid intrusion detection systems (IDSs) and security managers by recognizing application-level patterns of

usage that might indicate intrusions or security breaches and by gathering information useful to IDSs and security systems.

- *Integration and interfacing of multiple IDSs at the application level.* An application built within the QuO framework can easily interface to multiple mechanisms and managers, including multiple IDSs, through the QuO system condition interfaces.
- *Integration of IDSs and other resource managers.* The QuO toolkit provides support for building applications that can integrate interactions with managers for many different complementary dimensions (e.g., security, intrusion detection, and dependability) to achieve higher levels of service and adaptability.

The research performed within this project not only developed a powerful extension to the emerging area of distributed object middleware and a toolkit that supports it, but also demonstrated its use for an important class of problems in an important class of applications, namely the survivability concerns of critical applications. It also provided important experience about the nature of middleware adaptation, application assisted intrusion detection, and the feasibility of an approach to survivability that integrates together a number of more localized protection and security mechanisms to achieve more effective coverage.

Our research within this project also indicates natural directions for future research to build upon what we've accomplished. The adaptive middleware, languages, and abstractions that we developed have shown their utility to the extent that we've been able to test and demonstrate them. Further research is necessary to apply these to broader classes of attacks and other classes of applications, and to determine whether the abstractions we've developed are a complete and powerful enough set. Furthermore, while we've demonstrated the use of adaptation as a survivability technique, it would be useful to explore the nature of appropriate response and survivability strategies. That is, not only *whether* applications should adapt to survive, but *how* applications can adapt to survive certain classes of attacks. In addition, while we've explored the use of certain mechanisms, such as replication managers, IDSs, and access control systems, to aid in survivability, there are other mechanisms and managers that could be utilized. Some of these will be complementary, while others will conflict. The nature of using multiple managers and mechanisms and resolving conflict between them is an open area of interesting research. Finally, a natural open research topic that needs to be explored is the ramifications of using adaptation as a survivability mechanism. Adaptive applications, while being more difficult to attack, might also be more difficult for a system to manage. Furthermore, adaptation could be as powerful a technique for attacking or masking attacks as it is a technique for survival.

1. Goals of the Open Implementation Toolkit Project

More and more applications – critical and non-critical, military and commercial – are of increasing complexity and are deployed in wide-area, heterogeneous networked environments. Most of these distributed computing systems are already fragile: they break easily when run under conditions even slightly different from the environment in which they were tested. Many systems are so fragile that even foreseeable conditions, such as network congestion causing a remote service request to time out, can lead to system failure. This fragility has become a serious problem as our society increasingly relies on distributed systems for both military and commercial applications.

Most of these large scale, distributed systems are also increasingly vulnerable to attack. One reason is because they usually rely on a single implementation strategy whereas operating conditions are constantly changing. Another reason is that their distributed, networked nature provides more entry points for attackers than the closed controlled systems of previous years. The increased number of hosts, components, and hardware resources involved in distributed systems also increases the number of potential sources of failure. An increasing reliance on COTS products increases the possibility of attacks – both because the software is well known to attackers and because the software's black box nature can hide many vulnerabilities. Finally, as the vendors for a few products come to dominate the COTS market, the prevalence of those products and the resulting homogeneity in architectures, components, and applications increase the potential for attack and for propagation of attacks exploiting vulnerabilities of those products.

Rather than concentrating on point solutions, this project developed enabling technologies for developing *intrusion-aware, adaptable* applications, i.e., applications that can recognize intrusions, attacks, and malfunctions, and adapt to avoid them, thereby increasing their chances for survival. The Open Implementation Toolkit (OIT) supports the development of applications that can

- *Specify* their operating modes, needs, and normal ranges of behavior;
- *Measure* conditions in the system to determine whether they are operating within expected ranges;
- *Adapt* and reconfigure to recover from and avoid potential attacks; and
- *Control* available resources and mechanisms that aid in survivability, such as intrusion detection systems, security managers, and fault tolerance mechanisms.

2. Approach

The research in the OIT project built upon several emerging areas of research:

- Primary among these was research in Quality of Service (QoS) for distributed object systems undertaken by BBN in the Quorum (DARPA/ITO) AQuA and DIRM projects. Under these projects, BBN laid a basis for technology entitled *Quality Objects (QuO)* in which distributed applications could measure and control QoS aspects: availability in the case of AQuA and managed bandwidth in the case of DIRM.

- The concepts of *open implementation* influenced the OIT project. In open implementation, components of a distributed system are not treated as black boxes, as they historically are. Instead, information about their internal implementation and parameterized interfaces to influence their operation are made available.
- Finally, we were also influenced by emerging research in *aspect-oriented programming*, in which cross-cutting aspects of a program's operation are specified in special purpose languages and *weaved* into the application code by code generators.

Under this project, BBN developed a *toolkit* for the QuO framework that supports the development of adaptive, survivable distributed applications, i.e., applications that can recognize when they are operating outside acceptable ranges (indicating malfunctions or possible attacks) and adapt to avoid the problem areas. The toolkit allows an object programmer to develop distributed applications, objects, and subsystems with multiple implementations, each with distinct characteristics.

BBN's approach was as follows:

1) develop the components of the *QuO Open Implementation Toolkit*, including the following:

   - Description languages for specifying operating ranges, implementation alternatives, adaptation strategies, and the system resources and conditions that must be monitored at runtime to detect possible intrusions, malfunctions, and attacks; and code generators for producing adaptable applications from the descriptions.
   - A runtime kernel to monitor the appropriate system resources and conditions, recognize when an implementation is operating outside its acceptable range, and help the application reconfigure (e.g., change to an alternate implementation) to avoid the problem.
   - System condition objects that interface to property managers, mechanisms, and resources, such as intrusion detection systems and dependability managers.

2) Integrate with QuO components developed under other projects, including the following:

   - An ORB gateway shell, for supporting adaptation and control at the inter-ORB transport layer
   - Instrumentation support, including support for gathering information, inserting probes, and passing data along with round-trip method calls.

3) Demonstrate the use of the Open Implementation Toolkit by developing adaptive, survivable examples that interface to existing intrusion detection systems and property managers to receive information about potential intrusions and faults, and to provide guidance that will help the applications adapt for survivability.

3. Programmatics

3.1. Participants

BBN Technologies was the prime contractor on this effort, with the University of Illinois as subcontractor. The technical efforts were performed and managed within the Distributed Systems department of BBN. The principal investigator was Dr. Joseph Loyall, with significant contributions by Dr. Richard E. Schantz, Dr. Partha Pal, Dr. John Zinky, Rodrigo Vanegas, Michael Atighetchi, and James Megquier.

Dr. William Sanders was the principal investigator of the University of Illinois part of the effort, with major contributions by Dr. Michel Cukier, Jennifer Ren, and Paul Rubel.

3.2. Duration

The Open Implementation Toolkit for Creating Adaptable Distributed Applications project commenced July 15, 1997. It was originally scheduled to run until July 14, 2000. However, DARPA/ITO's Information Survivability program, under which the OIT project fell, was shortened, ending in early calendar year 2000.

3.3. Software deliveries

We delivered three versions of the OIT software to the Government. The first version was delivered to the Government on October 1, 1998 with source code, software, examples, and documentation. This version, version 1.0, contained the following features and capabilities:

- A QuO runtime kernel, including GUI visualization capability
- QuO code generators for Quality Description Languages (QDL), including Contract Definition Language (CDL) and Structure Definition Language (SDL) components
- Built-in instrumentation for QuO-enhanced CORBA applications
- Support for Java and C++ applications
- A library of sample and reusable QuO system condition objects
- Several application and QoS property examples that demonstrate how to build adaptable applications and QoS mechanisms using the QuO framework
- On-line documentation and how-to descriptions.

The second version of the OIT software was delivered to the Government on July 23, 1999 with source code, software, examples, and documentation. This version, version 2.0, contained all the functionality of version 1.0 plus the following improved and additional features and capabilities:

- The QuO toolkit completely supported both the Visibroker and TAO ORBs.
- Major improvements to QuO's Structure Description Language (SDL)
- A new QDL sublanguage, the Connection Setup Language (CSL)
- Powerful run-time instrumentation for system validation
- A variety of development environment and toolkit ease-of-use improvements
- Many application and QoS property examples that demonstrate how to build applications and QoS mechanisms using the QuO Toolkit
- On-line documentation and how-to descriptions.

The third version of the OIT software, version 2.1, was released on November 12, 1999 and will be delivered to the Government simultaneously with this Final Report, to coincide with the end of the Open Implementation Toolkit project.

The Open Implementation Toolkit version 2.1 contains all the functionality of version 2.0 plus the following improvements, features, and capabilities:

- Improvements to the QuO toolkit software, including to the QDL languages, instrumentation, and error reporting.
- Integration with TAO v1.0, the first commercially available version of the TAO ORB.
- A QuO gateway for integrating different transport layer protocols, mechanisms, and controls, plus an example gateway instantiation for the RSVP bandwidth management protocol.
- Support for survivability, using AQuA-based notification interfaces to recognize anomalous situations.
- Integration with Network Associates' Object-Oriented Domain Type Enforcement (OO-DTE) access-control software, providing a QuO-controlled security component.
- More examples, demonstrations, and documentation.

4. Adaptive open implementation toolkit

This section describes the components of the Open Implementation Toolkit.

4.1. Introduction and overview

The Open Implementation Toolkit is a fundamental piece of the Quality Objects (QuO) adaptive framework. The purpose of the toolkit is to support the development of applications that can *specify* the quality of service (QoS) needs, *measure* the QoS provided and available, *control* mechanisms and resources for providing QoS, and *adapt* to changing levels of QoS in the system.

QuO augments CORBA's functional view in a number of ways to support how the functionality is delivered, as well as to support the adaptive behavior mentioned above. The Open Implementation Toolkit provides the core capabilities of the QuO framework and consists of the following components:

- Quality Description Languages (QDL) for specifying operating ranges, implementation alternatives, adaptation strategies, and the system resources and conditions that must be monitored at runtime to detect possible intrusions, malfunctions, and attacks; and code generators for producing adaptable applications from the descriptions.
- A runtime kernel to monitor the appropriate system resources and conditions, recognize when an implementation is operating outside its acceptable range, and help the application reconfigure (e.g., change to an alternate implementation) to avoid the problem.
- System condition objects that interface to property managers, mechanisms, and resources, such as intrusion detection systems and dependability managers.
- An ORB gateway shell, for supporting adaptation and control at the inter-ORB transport layer
- Instrumentation support, including support for gathering information, inserting probes, and passing data along with round-trip method calls.

There are three complementary parts to QuO. The first part deals with the constructs needed to introduce the concepts for predictable, adaptable behavior into the application program development environment, including flexible specification of desired levels of QoS properties. The second part deals with providing runtime middleware to ensure appropriate behavior, including collecting information and coordinating any needed changes in behavior. The third part deals with the inserting the mechanisms for achieving and controlling each particular aspect of QoS that is to be managed, including aggregate allocation and control policies.

QuO's functional path is a superset of the CORBA functional path as illustrated in Figure 1. QuO interposes a *delegate* component in the client's functional path, whose purpose is to do the middleware-level QoS decision making. In a traditional CORBA application, a CORBA client would directly use the ORB to obtain an object reference and a CORBA server would create an implementation object and register it with the ORB. In a QuO application, QuO clients and servers create *connector* objects and call `connect()` on them. Each connector object is simply a delegate object (with the IDL interface of its remote object) that knows how to *connect* itself to the QuO infrastructure. As a result of a successful `connect()` call, the QuO application has a connected delegate. A QuO client then uses it as if it were the object reference, i.e., a *client-side delegate* provides the same interface as a client-side proxy or stub. A QuO servant uses it as if it were the implementation object, i.e., a *server-side delegate* provides the same functionality as the implementation object. The delegate and connector code is automatically generated by the QuO code generators based on the IDL and QDL specifications.

To use QuO, a client only has to modify the way it obtains the object reference, not all its invocations. Similarly, on the server side, only the creation and registration of the implementation object needs to be modified. Interposing the delegate in this relatively transparent manner allows the client's functional path to be instrumented and controlled in ways described below.

A QuO *contract* provides a means to specify what the client requires or desires in terms of QoS, as well as a means for it to be informed of what level of QoS it is actually receiving. This provides the basis for the client to cleanly specify, in an application-friendly manner, what to do when what it is actually receiving diverges from its expectations. QuO's Contract Description

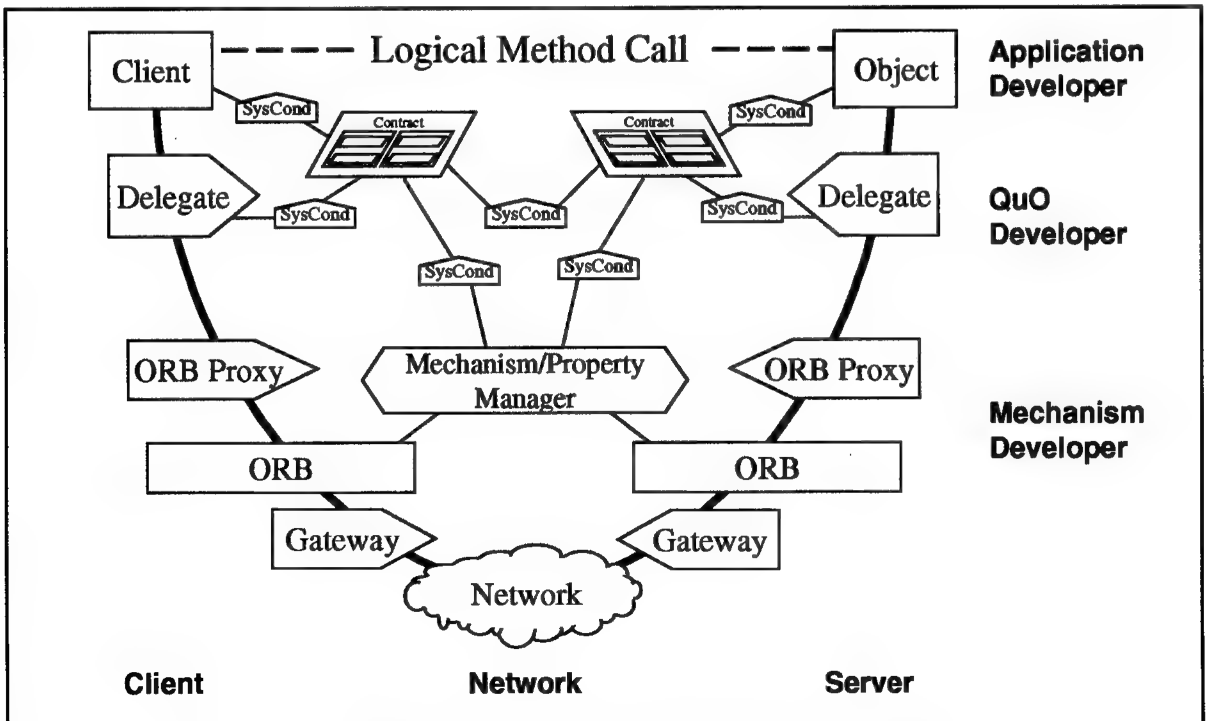

Figure 1: QuO Architectural Components

language is used for writing contracts, and is described in Section 4.2.1. The contract can specify *callbacks* to the client to alert it to when conditions have changed sufficiently to warrant the client being notified (and possibly adapting on its own behalf). QuO *system conditions* are objects that project a value into a contract. As such, they are a way for the contract to integrate information from different sources. Looked at from another point of view, they are the mechanism for connecting complex information sources into the QuO context in a relatively simple manner. System condition objects also provide a way for a contract to control or influence the way a property such as bandwidth or replication is managed, by serving as a conduit for passing on the client's requested level of service to the appropriate property manager.

Mechanism Managers (aka *Property Managers*) are responsible for managing a given QoS property (such as the availability property via replication management or the controlled throughput property via RSVP reservation management) for a set of QuO-enabled server objects on behalf of the QuO clients using those server objects.

The remainder of this Section describes the OIT components. Section 4.2 describes the Quality Description Languages and code generators. Section 4.3 describes the runtime kernel. Section 4.4 describes system condition objects. Section 4.5 describes QuO's instrumentation library. Finally, Section 4.6 describes QuO's object gateway.

4.2. Quality description languages (QDL)

QDL consists of three languages: CDL, which describes QuO contracts; SDL, which describes selection and adaptation information; and CSL, which provides for connector setup code generation. The following sections present an overview of these languages. More detail is available in Section 7 and in the following documents, available with the QuO Toolkit software:

- QuO Toolkit User's and Programmer's Guide
- QuO Contract Description Language (CDL) Reference Guide
- Structure Description Language (SDL) Reference Guide
- QuO Connector Setup Language (CSL) Reference Guide

4.2.1. The Contract Description Language (CDL)

CDL is a language for specifying quality of service (QoS) *contracts* in a distributed object computing (DOC) application, currently CORBA applications. Contracts specify the quality of service that is desired by the application, the anticipated range of service that might be encountered by the application, the system conditions that must be measured and controlled in order to achieve or recognize the appropriate QoS, and behavior to trigger when QoS levels change. Contracts can exist on the client side of an application, the server side, or both. QuO also supports multiple contracts, each of which can specify different dimensions of the QoS to be measured. For example, a secure, real-time application might use one contract to measure security in a system and another contract to measure the round-trip response time of the remote calls. Simultaneously, we are also exploring the issues of appropriately combining these separate views.

A QuO contract consists of the following components:

- A set of (potentially nested) operating regions, each representing a possible state of QoS. Each region has a predicate indicating whether it is active (the predicate is true) or not (the predicate is false).

```
contract InvContract( syscond  FileAddDeleteValueSC FileAddDeleteValueSCImpl fileAddedOrDeleted,
                      syscond IDSValueSC IDSValueSCImpl intruded,
                      callback  ClientCB clientCallback )
{
 syscond probe TimeProbeSCImpl OperationTime();

 region Normal ((OperationTime <= 500) and (not fileAddedOrDeleted) and (not intruded)) {}
 region TimeSuspect ((OperationTime > 500) and (not fileAddedOrDeleted) and (not intruded)) {}
 region AccessSuspect ((fileAddedOrDeleted and not intruded) or (not fileAddedOrDeleted and intruded)) {}
 region IntrusionLikely (intruded and fileAddedOrDeleted) {}

 transition any->TimeSuspect { asynchronous { clientCallBack.OperationTimeOut(); } }
 transition any->AccessSuspect { asynchronous { clientCallBack.ImproperAccessDetected(); } }
 transition any->Normal { asynchronous { clientCallBack.toNormal(); } }
 transition any->IntrusionLikely { asynchronous { clientCallBack.AccessDisabled(); } }
};
```

Figure 2: A contract to control and measure intrusion awareness

- Transitions, specifying behavior to trigger when the active region changes.
- References to system condition objects for measuring and controlling QoS. These are either passed in as parameters to the contract or declared local to the contract. System condition objects are used in the predicates of regions to measure values of system resources, object or client state, etc. and used in transitions to access QoS controls and mechanisms.
- Callback objects for notifying the client or object. Callbacks are passed in as parameters to the contract and are used in transitions.

The contract organizes the possible states of QoS, the information needed to monitor and control QoS, the actions to take when QoS changes, and the times at which information is available. The nesting of regions can be used to arrange regions according to logical groupings of information or time. For example, a contract can have a set of regions representing the QoS desired by an application, each of which has a set of nested regions representing the actual QoS that is observed. The outer regions would have predicates consisting of system condition objects that interface to the client and object and measure their desired or expected QoS. The nested regions would have predicates that consist of system condition objects that interface to and measure system resources. This grouping distinguishes the QoS associated with operating modes of the client and object, which will likely change infrequently, from the measured QoS of the system, which will probably change more frequently.

A programmer programs a contract class in CDL, similar to the way a C++ or Java programmer programs an object class instead of an object instance. At runtime, the connection routine (programmed using QuO's connection language, CSL) instantiates the contract objects and passes in the proper system condition objects and callback objects.

4.2.1.1 A Simple CDL Example Contract

The contract in Figure 2 specifies and controls intrusion awareness for an inventory application. The application has four operating modes of escalating suspicion that an intrusion is occurring. The `fileAddedOrDeleted` system condition object watches for unauthorized access of the file system, while the `intruded` system condition object interfaces to an intrusion detection system (IDS). Meanwhile, the `OperationTime` system condition object measures the time that inventory operations take.

As long as the file system monitor and the IDS are not reporting anomalies and operations are being serviced within a threshold defining reasonable behavior for these operations (500 ms), the contract is in the *Normal* region. When operations are taking too long (i.e., greater than 500 ms), but there are still no reported anomalies, the contract will enter the *TimeSuspect* region. When one, but not both, of the file monitor and the IDS monitor report an anomaly, the contract enters the *AccessSuspect* region. Finally, if both monitors report anomalies, meaning that an unauthorized file access has occurred and that the IDS has detected a potential intrusion, the contract enters the *IntrusionLikely* region.

Transitions between any of these regions will trigger a call to a method on the clientCallBack object, to make the application aware of the situation. The keyword **any** is used to indicate that the contract doesn't care in which region the transition originated. All of the client notifications are considered **asychronous** callbacks, i.e., a thread is spawned to make the call to the callback, so the transition behavior is not blocked waiting for a return.

4.2.2. The Structure Description Language (SDL)

SDL is a language for specifying the adaptive behavior possibilities and strategies for an adaptive QuO application. The basic structure of an SDL description is as follows:

```
delegate behavior for interface interface_name and contracts contract_list is
  delegate_variable_declarations
  adaptive_behavior_descriptions
  default_behavior_description
end delegate behavior;
```

The *interface_name* must be an interface defined in an IDL file being parsed on the same quogen run. The interface name can be fully qualified, i.e., if the interface *I* is defined within a module *M* then interface_name will be *M::I*. See the *QuO Connector Setup Language (CSL) Reference Guide* for further details on how to parse IDL files in a quogen run and how to instruct quogen to generate delegates for an interface.

The *contract_list* must be of the form *c1, c2,* ... where each *ck* is the name of a contract defined in a CDL file being parsed on the same quogen run. See the *QuO Connector Setup Language (CSL) Reference Guide* for further details on how to include CDL files to be parsed in a quogen run.

The *delegate_variable_declarations* section allows the programmer (i.e., the QoS developer) to specify variables that are initialized during the delegate creation and initialization time. These become member variables of the delegate class. Each *delegate_variable_declaration* specifies a local name to which it can be referred from within anywhere in the SDL file (i.e., the scope of these variables are the whole SDL file).

The *adaptive_behavior_descriptions* section contains a list of method calls and/or returns for which the delegate defines alternative behaviors. Each of these in turn specifies a list of contract regions and behavior descriptions that, if the contract region is active when the method is called and/or returned from, will cause the delegate to execute the corresponding behavior.

The *default_behavior_description* allows the specification of behavior to be executed when any method that is not specified in the SDL description is called.

More detail about the syntax and semantics of each of these sections is contained in Section 7.

4.2.3. The Connector Setup Language (CSL)

CSL is a language for specifying the components comprising a QuO application, how they are instantiated, initialized, and hooked together to produce a distributed application with QoS measurement, control, and adaptation. CSL specifies the following:

- The IDL, CDL, and SDL files that must be processed to produce an application,
- The CORBA objects, system condition objects, contracts, and delegates that need to be instantiated
- The way in which the various objects are created and initialized
- The relationships between the objects, e.g., the system condition objects used by each contract, the remote objects wrapped by delegates, and the nesting of contracts or delegates
- Any initialization code that should be performed at application setup time.

Executing the code generator on a CSL file triggers the parsing of the various IDL, CDL, and SDL files, similar to a Unix Makefile. It also generates a single Connector class and methods that can be called in the application code in place of calls to instantiate or locate the individual remote object references. CSL allows the QuO programmer to specify arguments that the application will pass into the connector, such as references to an ORB, BOA, or application-specified initialization data.

A QoS programmer programs a connector class in CSL, similar to the way a C++ or Java programmer writes a Makefile. The code generator parses the CSL file, triggers parsing of all the IDL, CDL, and SDL files, and generates code for a Connector class that, when instantiated, creates the relevant objects, connects them, and invokes any relevant initialization code.

The example CSL description in Figure 3 comes from one of the example applications that ships with the QuO toolkit, **simple**. **Simple**, as its name suggests, is a simple example of the use of QuO in an application. It consists of a single client and a single server. The server for this application maintains one piece of data, an integer variable. The `count` method adds an integer, passed as a parameter, to the value at the server and the `countDown` method subtracts the integer from the server's value. There is a single system condition that can be changed by external conditions (such as user intervention) and controls whether the application wants to increment or decrement the counter. The contract records whether the system (as controlled by the system condition object) is in a state where the client's calls should increment the server, decrement the server, or do nothing.

The first section of the CSL code, **Connector Attribute Specifications**, describes programmer specified attributes about the generated Connector code. The **class** specification specifies the name of the generated connector class, while the **interface** specification specifies the name of the superclass of the generated connector class. **Codeloglevel** is a debugging flag, specifying how many or how few messages you want to see in the generated code. The value for **codeloglevel** can be "**none**", "**low**", or "**high**".

Cppinclude is a line to allow you to specify header files to be placed in a **#include** statement at the top of the generated connector code (if C++ is being generated). Multiple files are separated by spaces. Finally, **target** is a way to specify whether connector code is being generated for the **client** side or the **server** side.

The next section, the **Include section**, specifies the files that comprise a QuO application and, therefore, need to be processed by the QuO code generator. These **include** statements drive the

```
/* This is the CSL specification used to setup the connector for simple  */

/* ----------------- Connector Attribute Specifications -----------------*/
class                "SimpleConnector"
interface            "SimpleConnectorInterface"
codeloglevel         "high"
cppinclude           "CounterCallbackImpl.hh"
target               "client"
/* -----------------Include section------------------------------------- */

/* Include cdl files */
include "qdl/CounterContract.cdl"

/* Include sdl files */
include "qdl/CounterDelegate.sdl"

/* Include idl files */
include "idl/Counter.idl"
include "idl/CounterCallback.idl"

/* ----------------Object Definition section---------------------------- */

/* Define Remote Objects */
Counter thisRemoteObj = fileior ("Counter.ior") ;

/* Instantiate System Condition Objects */
ValueSC countDirection = new ValueSCImpl ( "Count Direction" ,
                                            "com.bbn.quo.ValueSC" ,
                                            "com.bbn.quo.ValueSCImpl" ) ;

/* Instantiate Callback Objects */
CounterCallback theCallback =
  new CounterCallbackImpl ( "Count Direction Callback" ) ;

/* Instantiate Contracts */
quo::Contract contract1 =
  new CounterContract ( "Counter Contract" ,
                        "com.bbn.quo.examples.simple.CounterContract" ,
                        countDirection , theCallback ) ;

/* Instantiate Delegates */
returndelegate thisDelegate ( thisRemoteObj , contract1 ) ;

/*------------------Function Calls ----------------------------------- */
dumpior countDirection "CountDirection.ior" ;
```

Figure 3: Example CSL specification for an application using QuO

QuO QDL parsing. All the CDL files, SDL files, and IDL files that need to be processed to generate the contract, delegate, and connector code must be specified in this section. Their pathnames must be specified relative to the directory in which **quogen** is invoked. In this example, the simple example includes one CDL file (**qdl/CounterContract.cdl**), one SDL file (**qdl/CounterDelegate.sdl**), and two IDL files (**idl/Counter.idl** and **idl/CounterCallback.idl**).

The third section, the **Object Definition Section**, is the section in which you specify the remote objects that need to be instantiated, how they are initialized, and how they are hooked together. The first statement in this section specifies that an object of type Counter is to be located by reading its CORBA IOR from the file **Counter.ior**. The local variable **thisRemoteObj** will refer to the object during the remainder of the CSL file.

The next statement creates a system condition object called **CountDirection**. **ValueSC** and **ValueSCImpl** are an IDL type and a Java class, respectively, provided by the QuO libraries. The next statement creates a callback object. **CounterCallback** and **CounterCallbackImpl** are an IDL type and a Java class specific to this application.

The next statement describes the QuO contract used in this example. All contracts are of type **quo::Contract** but are instantiated using the class name specified in the CDL file, in this case **CounterContract**. A contract instantiation always takes two strings as arguments, plus the system condition objects and callback objects as described in the CDL code. The first argument identifies the contract as named **"Counter Contract"**. The second argument identifies the Java class of the contract. The third and fourth arguments hook the system condition object, **countDirection**, and the callback object, **theCallback**, which we created above, into the contract.

The final statements of the **Object Definition Section** identify the delegates that need to be created, what contracts and objects they wrap, and the delegate that is the top most wrapper around the remote object's stub. In this example, we create a single delegate that points to the **thisRemoteObj** remote object reference and uses the **contract1** contract.

The final section of the CSL code is the **Function Calls** section. This allows you to specify a set of function or method calls that need to be made as part of the connector code once all the objects are created and instantiated. This gives the opportunity to perform some initialization on the objects as part of the connector code. In this example, a call to the CSL primitive **dumpior** is made to write the IOR of one of the created objects, the **countDirection** system condition object, to a file. This is so it is accessible outside the example application code.

4.3. Runtime kernel

The QuO kernel is a library of services, implemented in Java, providing the basis for the QuO runtime system, contracts, and system condition objects. The kernel library provides two essential services: an object *factory* and a *contract evaluator*.

The object factory creates and initializes contract and system condition objects. Currently all of these objects are created at the startup of a QuO application, although future QuO applications might require dynamically created contracts or system condition objects. The factory is implemented using reflection in Java, since it is difficult to anticipate at compile time the classes of contract or system condition objects that will be used by the runtime system and how many instances will be instantiated by the delegates. A delegate makes a CORBA call to the QuO kernel factory to instantiate an object, i.e., either a contract instance or a system condition object, passing the class of the object as a string. The factory locates the corresponding class file and loads it if it has not already been loaded. The factory then creates and initializes an instance of the class and returns the object reference to the delegate. The kernel maintains a record of the objects used by each delegate, so that it can clean up when applications quit or delegates disappear.

The contract evaluator schedules the evaluation of contract objects in the QuO runtime for a given application. A QuO application might involve any number of contract objects, whose evaluation can be triggered in the two following ways:

- A delegate's *premethod* or *postmethod* QoS check – When a client makes a remote method call, the call goes to the appropriate delegate. The delegate checks the current state of QoS (as represented by the current regions of a contract) to decide what adaptive behavior to select. Likewise, the delegate will often check the state of the contract upon a method's return, to decide what to do with the return value. Each of these trigger contract evaluation.

- A change in an observed system condition – Some of the system condition objects that a contract uses to collect data about QoS in the system are *observed* by the contract (as specified in QDL). When the value of any observed system condition object changes, it triggers contract evaluation.

Even though there might be many contracts concurrently ready for evaluation, we expect the QuO kernel to consume a small overhead of CPU cycles relative to the application, therefore QuO does not concurrently evaluate contracts. Instead, triggering simply marks a contract object as ready for evaluation. The contract evaluator is a single high-priority thread that performs a round-robin check of all contract objects in the runtime. If a given contract object is marked, the evaluator evaluates it before proceeding to the next contract. If evaluation of a particular contract is triggered more than once before the evaluator gets to it, the contract is only evaluated once.

The QuO kernel is flexible about its environment. It can run in its own process or in the same process as the application. It can run on the same or a different host as that of the application. Finally, any number of applications can share it whether or not they are on the same host. The kernel also includes a GUI for displaying the current state of contracts and system condition objects.

4.4. System Condition Objects

Every system condition object represents a single system property value, such as time elapsed since the last invocation or number of active jobs on a remote host. Contract objects use these values to determine the current operating region. In addition, arbitrary functionality can be inserted into the implementation of system condition objects such as control of mechanisms external to the QuO kernel or internal logic required to compute the value which the system condition object represents. This functionality can execute asynchronously to the QuO runtime and contracts or it can be triggered by contract transitions.

Each system condition object is implemented as a Java class which *implements* a simple Java *interface* common to all system condition objects. This interface must provide a *getValue()* method used by contract objects to query system condition objects for their values and a *get-<type>()* method for any type relevant for the system condition object. Some simple system condition objects, e.g., those that are internal to a contract, are completely specified if they implement the getValue() method. All other system condition objects must be CORBA objects and as such must inherit from an existing hierarchy of system condition CORBA objects. This requirement makes it possible for the delegate to instantiate the system condition objects (using the QuO kernel's factory).

Since system condition objects can publish their IDL interface, they can provide control and access to their values outside the QuO runtime, effectively allowing system condition objects to function as interfaces between QuO and system resources, mechanisms, managers, etc. external to QuO. Each system condition object may also run any number of threads exclusive to either each instance or each class. The freedom to implement system condition values with asynchronous control does not interfere with the general control flow within the kernel and facilitates the implementation of certain kinds of objects such as timers or resource pollers. A growing library of system condition classes that comes with each QuO release and that can be composed or inherited from also facilitates the implementation of new system condition objects.

The following examples give a sense of the expected granularity of system condition objects. Of the five system condition classes that follow, the first four are part of the standard system condition library and the fifth can be trivially implemented in terms of the first.

- **ValueSysCond** is a simple system condition object representing a value holder. Its getValue() implementation simply returns its state, and its state is set (and read) externally by the methods writeVal() and readVal() defined in the class' IDL interface and implemented as one would implement the methods of any CORBA object.
- **MethodInvocationTimerSysCond** is a more complicated system condition object implemented in terms of ProbeSysCond and TimerSysCond. The ProbeSysCond class provides the object with signals that indicate when a method invocation has begun and when it is about to end. The TimerSysCond class provides a generic implementation of a timer.
- **RemoteSystemLoadSysCond** gets its value by periodically polling a remote host for its load statistics.
- **RSVP_SysCond** has a getValue() method that simply reports the current status of a particular RSVP connection. However, embedded in the implementation of this system condition object are the control mechanisms required to build up and tear down the RSVP connection whose status it is reporting.
- **RemoteObjectStateNoticeSysCond** reports when the state of a remote object last changed. Rather than implement this by polling, the remote object can signal this system condition object when its state changes.

4.5. Instrumentation library

The QuO toolkit supports making in-band timing measurements of each CORBA call. For each call, a trace-record is generated which records the amount of time the CORBA call spent in the network, the object server, and QuO delegates. In addition, the trace-record records the size of the input and output parameters. The trace-records are given to a statistical reduction object which uses the per-call measurements to calculate real-time statistics, such as Average Call Latency, Network Capacity, Object Capacity and QuO latency overhead. The real-time statistics are passed to QuO system condition objects and can be used to define regions of a QuO Contract.

QuO instrumentation is featured in the QuO examples found in `examples/bottleneck` and `dirm/examples/bottleneck` in the QuO release software. These examples demonstrate how to use the instrumentation to detect a performance bottleneck in real time.

4.5.1. Components of QuO Instrumentation

The in-band instrumentation is extremely flexible and depends heavily on QuO SDL to place the probes. QuO instrumentation consists of the following components:

- **StandardInstrumentation**: a Java object that contains the methods that are to probe the QuO delegate into a trace-record, collect real-time statistics, and convert the statistics to a signal to be sent to system condition objects. An instance of a `StandardInstrumentation` object is attached to each delegate. The QuO V2.1 Javadoc documents the `StandardInstrumentation` methods.

- **RunLine**: a simple Java object that does linear regression on a time series. The QuO V2.1 Javadoc documents the `RunLine` methods.
- **PropertyProbe system condition object**: this system condition object receives the real-time statistics from the StandardInstrumentation object in the Delegate. The signal is used to transfer the information each time the delegate requests the contract to be evaluated (twice per call).
- **Modified Functional IDL**: used to create a channel to move the trace-record from the client-side delegate to the server-side delegate. The modification is to add an extra parameter to each method signature, which the client-side adds and the server-side strips off.
- **Server-Side Delegate**: used to collect measurements on the server-side and return the trace-record back to the client-side. The server-side delegate is actually a CORBA object with an IDL that includes the extra parameters.

QuO SDL is used to place the `StandardInstrumentation` probes in the generated delegate code.

4.6. Pluggable gateway

In order to provide the type of controllable, predictable, and manageable environment we seek with QuO, we need mechanisms to control and enforce resource management and synchronization for the networked entities. An important factor behind the rising level of interest in QoS is the increased embedded use of still largely unmanaged network communication services. The inherent variability in using these services, due to changes in resource configuration, load, relative location and current availability (operational status), make the end to end results delivered to the application highly unpredictable. This has led to the development of a variety of mechanisms and approaches for better manageability. However, these mechanisms are usually at a fairly low level in the protocol stack or closely tied to the transmission mechanisms where they can better control behavior. That makes them difficult to incorporate into the software engineering paradigms close to the application programmer's level of abstraction. One possible approach is to develop an enhanced specialized ORB with just the right properties. A better alternative, which we are seeking, would work with a variety of off-the-shelf ORB products, both commercial and experimental.

Our solution, and the one adopted for QuO, is to combine control elements at the object interface level with control elements at the transport level in a translucent manner. That is, to link the desired behavior at the client/object interfaces with the appropriate behavior at the communication interfaces, in a manner which makes the connection between the two visible and controllable (versus transparent). An important component supporting the integrated property QuO architecture is the *QuO object gateway*. An object gateway is the QoS aware element inserted at the transport layer between clients and objects to provide the managed communication behavior for the particular QoS property being supported. In the QuO architecture, it is the job of the QuO object gateway superimposed at the transport level to apply the appropriate mechanisms needed to fulfill the obligations incurred at the QuO contract level. Figure 4 illustrates the QuO Object Gateway.

Since it is highly desirable that the QuO system be useable with a variety of ORB products, and the ORB is generally responsible for establishing communication with the designated object, our immediate goal becomes inserting the proper mechanism after ORB processing, but before handing the request off to the network transport subsystem. In CORBA, the open protocols be-

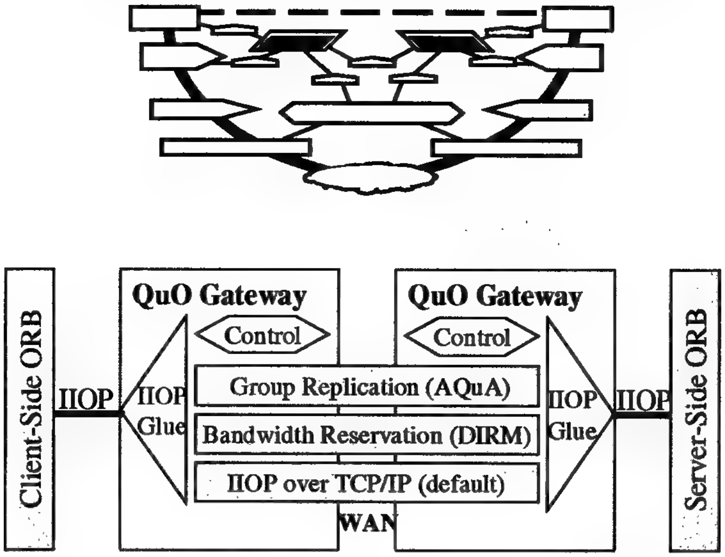

Figure 4: The QuO gateway

tween ORBs needed to support ORB interoperability (Interoperable Internet Operation Protocol, or IIOP) also allows the seamless insertion of a QuO gateway function to provide the appropriate transport level conditioning needed to meet the high level QoS contract.

4.6.1. Functions of the Object Gateway

The primary function of the QuO Gateway is to allow the easy and convenient insertion of QoS aware transport layer protocols between distributed clients and servers. By transport layer protocols we include not only traditional protocols (e.g., TCP/IP) for moving data but also specialized protocols to support specific QoS enhanced data transport mechanisms providing specific attributes in the areas of real time performance, dependability and security. These specialized protocols contribute various enhanced properties to the transport of message data, ranging from reserving bandwidth capacity to ensure high priority, real time message data traverses the network unimpeded by further delay from other competing traffic, to organized group (multicast) transport distribution to ensure synchronized parallel, redundant transformation updates, to controlling which messages can or can't get through and with or without proper encoding to support overall security objectives.

In its CORBA instantiation, the QuO Gateway piggybacks on the existing IIOP invocation/response transport mechanisms, inserting additional protocols that manage QoS properties as noted above. In effect, the QuO Gateway serves to manage enhanced IIOP transport interactions between CORBA clients and objects. While transport level QoS is not equivalent to end-to-end QoS, it does nonetheless address a significant, and perhaps most critical, part of the end-to-end problem. Another step in QoS management is to link the transport level resource management provided by the QuO gateway with other parts of the QuO framework to produce the desired managed, adaptive and integrated behavior.

The functions of the QuO gateway fall into two distinct categories: (1) those that are needed for the general QuO gateway support independent of the property being managed and (2) those that are specific to the various QoS property protocol mechanisms being inserted into the general

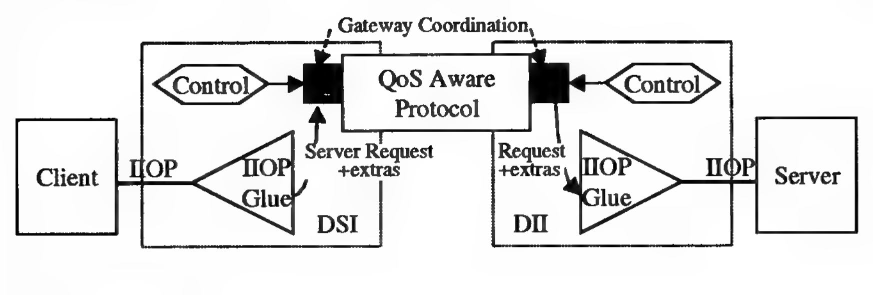

Figure 5: Object Gateway Shell Functional Overview

gateway structure. In the rest of this subsection we enumerate the general gateway functions supported.

The standard functions of the object gateway include the following:

- (Approximate) transparent insertion in the IIOP stream of QuO specific management functions (capable of calling appropriate property specific protocols) on both the client and server sides if needed. This involves inserting in the IIOP stream matching gateway halves providing a *shadow* QoS transport server on the client end, and a *shadow* QoS transport client on the server end. In this way, to the client the QuO gateway looks like an endpoint CORBA server, while to the server, the other half of the QuO gateway looks like an initiating CORBA client. The job of the gateway is to transport the request/reply to its appropriate destinations utilizing the appropriate path selection and mix of QoS mechanisms.
- Request/Reply matching needed to coordinate the two asynchronous activities
- Error handling, including system exceptions acting on behalf of the ORB, as well as other CORBA based redirection functionality such as "Cancel" and "Locate".
- Standard interfaces for inserting and substituting a wide range of specialized transport level QoS mechanisms and transport level QoS handlers in support of these mechanisms.

4.6.2. Architecture, Design, and Implementation of the Object Gateway

The gateway shell provides basic gateway capabilities, based on monitoring IIOP requests. The gateway inserts itself into the inter-ORB communications stream by terminating the IIOP session at the client side and initiating a new one at the remote site. In between, it applies appropriate QoS measures depending on the current contract region and available mechanisms and resources. This gateway shell is suitable for use as the base for constructing specialized QoS-property object gateways.

A specific property object gateway is built by layering the gateway shell on top of underlying QoS implementation mechanisms. For example, we layer the QuO object gateway over QoSME to produce a RSVP gateway. The gateway translates from IIOP to specific requests on the underlying RSVP mechanisms. Similarly, we layer the gateway over Ensemble group communication to provide synchronized message traffic among collections of replicated objects. The gateway translates from the single object invocation to the underlying group communication transport paradigm. Figure 5 illustrates the general architecture of the QuO object gateway shell, including the interaction of its two gateway halves.

To the client ORB, the QuO gateway looks like the object. To the server ORB, the QuO gateway looks like a client. The ends of the gateway are at minimum on the same LAN as the

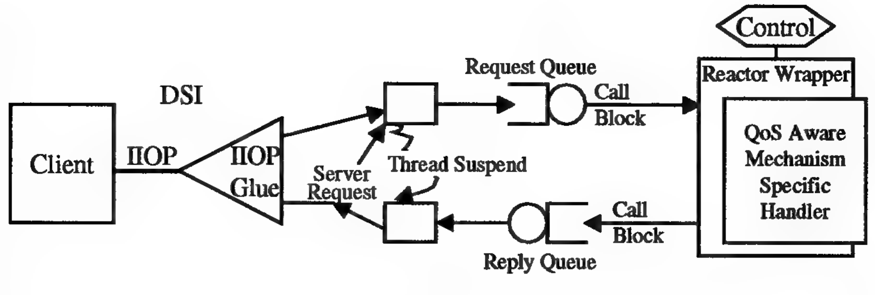

Figure 6: Client side gateway shell

client/object, and may be on the same host. In certain configurations, there may be many gateway instances on a LAN, and even many gateways on a host. CORBA objects are used to control QuO gateway halves by allowing the gateway to be configured remotely, but do not interfere with in-band communication.

DII and DSI are standard CORBA interfaces provided to support dynamic invocation semantics, where the nature of the parameters of the invocation/reply is not known until runtime. We use them to insert our QuO gateway proxies as object request relays, utilizing the CORBA standard GIOP messages underlying the IIOP implementation. IIOP glue multiplexes and demultiplexes GIOP messages to the gateway coordinator. The Gateway Coordination module handles flow control and error detection, as well as path selection.

Figures 6 and 7 show the symmetry in the gateway shell halves representing the client side gateway and the server side gateway. Transport interactions between these halves are governed by the appropriate mechanism for the specific QoS property being managed by the gateway within the QuO infrastructure.

The client-side interface produces a partially parsed GIOP request to be transported that is bundled into a call block and puts it on the request queue for the specialized transport. The call block is used for sequencing the transport activities and for matching with an eventual reply. In our current design, there is a separate thread per IIOP transaction, with the suspended processing thread maintained in the call block.

The server-side interface essentially does the inverse of the client-side, starting from the transported GIOP request. The DII interface sends a request structure to the object reference. The return values are bundled into the call block and put on the reply queue for the specialized transport. Reply messages are forwarded with very little change and are largely opaque to the gateway.

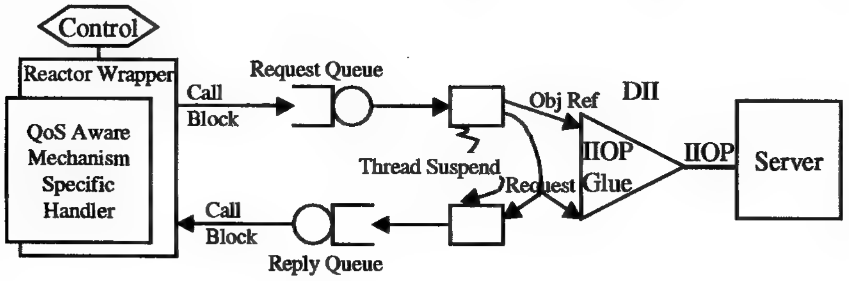

Figure 7: Server side gateway shell

If the transported call encounters an error, an exception is thrown in the reply code. These exceptions are classified as System_Exceptions, because the gateway is logically an extension of the ORB. Special threading restrictions are needed to maintain the order of arrival of requests, which are important to some QoS mechanisms.

5. Survivability using the QuO Toolkit

Using the QuO toolkit, described in Section 4, we support the following system level behaviors to improve the survivability of applications:

- *The development of intrusion- and security-aware applications.* These applications can aid IDSs and security managers, by recognizing application-level patterns of usage that might indicate intrusions or security breaches. QuO instrumentation (Section 4.5) and system condition objects (Section 4.4) measure the level of service provided. These can also gather information useful to IDSs and security systems, both for recognizing intrusions and for gathering information about their causes and sources.
- *The development of survivable applications.* These applications can adapt to changing conditions in their environment, including reported intrusions and changes in security policies. This enables them to avoid potential intrusions, continue in the face of degraded service, and recover from intrusions and faults.
- *Integration and interfacing of multiple IDSs at the application level.* While many IDSs are good at detecting certain types of intrusions, a 1998 DARPA ISO evaluation showed that multiple IDSs cover a larger space of potential intrusions [Dyn98]. However, most IDSs are not designed to work in conjunction with others. An application built within the QuO framework can interface to multiple mechanisms and managers, including multiple IDSs. QuO's system condition objects (Section 4.4) provide a common interface to mechanisms and managers that have proprietary interfaces. In this manner, QuO applications can access information from multiple IDSs that detect different types of intrusions. Let us emphasize that *the idea of interfacing uniformly with multiple IDSs at the application level is not to come up with a better IDS, rather to increase the security coverage to the application.* This is complementary to the Common Intrusion Detection Framework (CIDF) effort [Sta98], which is developing a framework for IDS-to-IDS communication with an aim to perfect the art of intrusion detection. CIDF does not provide any support for application-IDS cooperation.
- *Integration of IDSs and other resource managers.* IDSs and other managers, such as security policy managers or dependability managers, perform complementary activities and could cooperate to provide higher levels of service. For example, a security policy manager could use intrusion detection information provided by an IDS to dynamically determine whether to move to a stricter level of access controL Likewise, an IDS could use information from a fault detection or dependability manager to determine where and what type of intrusions to look for. QuO provides support for building applications that can integrate interactions with managers for many different complementary dimensions (e.g., security, intrusion detection, and dependability) to achieve higher levels of service and adaptability.

QuO supports the addition of security awareness and control at the middleware level, i.e., at the interface between applications and the lower level security and resource management mechanisms. We concentrate on middleware capabilities for the following reasons:

- Survivability mechanisms at the middleware layer can be more effective at application survivability because of their closer proximity to application information. OS and network level resources and mechanisms can be crude with regard to detecting whether an attacker has corrupted an application (e.g., recognizing whether processes have failed or whether an application is requesting resources). In contrast, survivability mechanisms in middleware can better detect certain attacks on applications, e.g., detecting whether an application has violated any of its constraints, and use OS and network level resources and mechanisms in an application-specific way to combat them.
- Mechanisms at the middleware layer can be uniform across heterogeneous operating systems and networks. The application can use the same approaches to security and adaptation on NT as on Unix, and can continue using them even if migrated from one kind of host to another. Furthermore, uniform policies and response strategies can be applied across the pieces of heterogeneous distributed applications.
- Security mechanisms in current operating systems and networks are far from perfect. This is partially the result of the increasing use of COTS software, not designed with security in mind, and partially the result of the limited scope and knowledge of the application available at the network and OS levels. So the survivability of host systems needs to be enhanced, and a natural place to do that is in the middleware that ties the host systems together. We use QuO to *enhance* the survivability of applications and host systems, working in conjunction with security mechanisms at lower levels. If security in the operating systems that host QuO can be circumvented, then QuO's adaptability is also in danger of compromise on those hosts.

In the following sections, we discuss in more detail two aspects of middleware enabled survivability: *survival by adaptation* and *survival by protection.* The survivable example applications developed under the Open Implementation Toolkit project concentrate on *survival by adaptation*, but laid the groundwork for pursuing and demonstrating *survival by protection* under other projects.

5.1. Survival By Adaptation

Faced with changing environmental or operating conditions, an application may:

- do nothing,
- try to isolate itself from the changes,
- try to anticipate the possible changes and include code to handle each possibility, or
- employ general purpose mechanisms to adapt to changes.

Obviously, the first option can lead to application failure when changes (malicious or otherwise) occur. Regardless, the first option and ad hoc attempts to encode the second and third options are the most common approaches in use today. A more systematic, software engineering approach to the fourth option, which encompasses useful subsets of the second and third approaches, is the underlying theme of our approach.

5.1.1. Adaptation

Let us start with several adaptive strategies that could help an application survive. Consider a simplified application where the client requests images from an image server. For this application, assume that receipt of images in a timely fashion is more important than whether the image is large or small, processed or unprocessed. In other words, timeliness (when the image arrives) is more important than the precision (size of the image) or accuracy (quality of the image) of the image. Without any adaptive behavior, the application may cease to be useful (or even crash) if the network or the server slows down. However, with adaptivity we can craft the following survivability scenarios:

- If a denial of service (DOS) attack on the server causes its CPU to overload, thereby slowing image delivery, the image server starts sending unprocessed images. This saves the CPU time that would have been used to process the images and sacrifices data quality for performance.
- If a DOS attack floods the network with traffic, the image server starts sending smaller images to save on the network overhead associated with larger data sets, thereby trading off quantity of data for performance.

Note the important role played by monitoring the environment. In order to take a meaningful adaptive step towards survivability, the application needs to know whether the problem was in the network or at the server host (because the remedy will be different). Also note that without the adaptive response, merely knowing the existence or the source of the problem does not ensure the application's survivability. *It is easy and straightforward under the QuO framework to formulate the operating regions and the adaptive behaviors of such an application and implement the contracts and the delegates and insert the monitoring mechanisms in the application using the QuO toolkit.* The Bottleneck example that is included in the QuO software illustrates this.

Two comments about such an example are in order. First, it demonstrates the QuO toolkit as an enabling technology toward the feasible development of applications that can more easily adapt to their environment. Second, it shows that this kind of adaptation can be effectively applied to ensuring the survival of applications as defined earlier, i.e., to avoid potential intrusions, continue in the face of degraded service, and recover from intrusions and faults.

5.1.2. Redundancy

Systems with redundancy offer the possibility of increased survival. Redundant hardware is usually the primary route taken to ensure survivability of physical systems. The same is true in the context of software systems as well. An application can utilize redundant sites (with reserved bandwidth on each path) for the same service so that if problems make one path unusable it can still use the service via the other. It may employ redundant servers so that if one crashes the others would still service client requests. If off-line reconfiguration and restart is not a useful primary option, then effective use of redundancy involves elements of (online) monitoring and adaptation. For instance, the middleware needs to know about a server crash so that it could dispatch the next request to an alternate server. Traditionally, this is either done by the application code (e.g., time out and then redirect to the alternate server) or by some kind of redundancy management system, which is often coupled with a specific resource.

Using QuO adaptive behavior descriptions it is easy to redirect invocations to an alternate server. Using QuO's integration with property managers, we can support an even more sophisticated kind of redundancy. For instance, to survive crash failures we can use the Proteus dependability manager [Cuk98, Sab99] which supports various replication strategies including active replication, where it manages multiple replicas of an object. Without any kind of adaptation, merely by asking Proteus to maintain two replicas on two hosts, we can ensure that the application will survive the crash of one replica or one of the hosts, and Proteus will try to start another replica when such a crash happens. With QuO, we can define an even more sophisticated adaptive strategy for survivability: we can define contract regions that signify whether or not the dependability mechanism can sustain the desired number of replicas. Then when the application senses that the dependability mechanism cannot sustain the desired number of replicas, it may withhold operations that are known to cause problems at the server (for instance, new methods that may not be fully tested and debugged as yet) or suppress all low priority invocations. Similarly, using QuO and the RSVP bandwidth management mechanism [BBN98], instead of reserving bandwidth a-priori, a QuO application can reserve a channel as part of its adaptive response when it senses that available bandwidth is less than what is expected (which will result in less bandwidth for other, less critical, applications).

5.1.3. Monitoring

In the preceding two sections we have stressed the integral role played by monitoring in survival by adaptation. Monitoring is important because it provides the awareness of the surrounding environment and the events that are taking place in the environment, which is a prerequisite for intelligent adaptive decision making. In addition to measuring systems resources such as the available bandwidth, CPU load or number of sustainable replicas, monitoring may include interfacing with devices and mechanisms such as firewalls and intrusion detection systems. With sophisticated monitoring it is possible to construct advanced survivability scenarios such as the one described below.

Imagine an application using replicated objects for survivability. It has requested enough replication to tolerate 2 crash failures and accordingly the Proteus dependability manager is maintaining 3 replicas on 3 different hosts. Although Proteus is able to successfully maintain the desired level of replication, replicas on one of the hosts are crashing frequently. A QuO system condition object can recognize the patterns of failures on that host and as a result, the application (as part of its adaptive response) can request an intrusion detection system to monitor the host more carefully. This activity may detect an act of intrusion and with this information from the monitoring mechanism, the application can further adapt by asking the Proteus dependability manager not to place replicas on that host.

5.1.4. Diversity

Diversity has a natural appeal for survivability. In the context of survivable applications diversity can exist at various levels, e.g., different hardware and operating system platforms, different algorithms and implementations, and different intrusion detection mechanisms. By definition, diversity entails some form of redundancy as well. Sometimes diversity provides better coverage or improves accuracy. For example, by combining one intrusion detection system that focuses on the network interface with another intrusion detection system that focuses on the file system one can detect more kinds of intrusion attacks (better coverage). For those attacks that affect or involve both network and file system resources, such a combination may also reduce false alarms (improve accuracy). For other cases diversity provides a resistance that homogeneity cannot provide: the vulnerability of one operating system may partially be compensated by using two operating systems since the risks are reduced that the same attack will bring down both of them. Using diversity, applications can employ sophisticated survivability scenarios as described below.

Continuing with the applications that use replicated objects, let us consider that the dependability manager can manage replicas on two different operating systems, NT and Linux. Imagine that the monitoring mechanisms indicate that an NT attack is in progress. As part of its adaptive response the application can ask the dependability manager to remove all replicas from NT hosts and instead use those hosted on Linux boxes (Having replicas on multiple operating systems may be slightly more complex if the implementation of the replicated object is not portable.). Furthermore, when the monitoring mechanism senses that replicas are crashing more frequently on a host, it may employ more than one IDS to watch over the host: one monitoring the file system integrity and the other monitoring network activities.

5.2. *Survival By Protection*

The first line of security defense for any application under the QuO framework is access control The application designer needs to ensure that only authorized users and programs may invoke the application and that no tampering with the application's internal mechanisms is possible. QuO integrates access control mechanism technology, in the form of Network Associates' OO-DTE (Object Oriented Domain Type Enforcement), to satisfy these needs. We assume that the environment in which QuO runs, including the host operating systems and networks, has built-in security mechanisms. Without such protection, QuO's security mechanisms can be easily disabled. We do not assume, however, that the environment offers uncircumventable security.

5.2.1. Object Oriented Domain Type Enforcement (OO DTE)

OO-DTE [Ste99] is an object-oriented, policy-driven mechanism for fine-grained access control in distributed systems. It is policy-driven because it bases access control decisions on a single, explicit, written policy governing an entire distributed application. The application developer describes the access controls once, and OO-DTE enforces these controls consistently at all locations where the application runs. This approach eliminates the need for a developer to set operating system access controls manually on every host.

OO-DTE is object-oriented because it controls access in terms of objects and the clients that use them. It works at the middleware level, above the level of host access controls on resources, which are represented to the designer as abstract objects. OO-DTE access control is fine-grained because it allows control over access to each object and each object method individually. The protection it offers is therefore more flexible than that offered by firewalls, for example. OO-DTE policies are written in a language called DTEL++ that defines a set of access rights for the application. DTEL++ relates these access rights to elements of CORBA. Objects are assigned rights to access methods declared in CORBA IDL or rights to access objects by name, using the CORBA Naming Service. QuO application developers use OO-DTE through DTEL++.

5.2.2. Using OO-DTE as a Security Mechanism in QuO

QuO application developers can use the OO-DTE mechanisms directly for protection. The developer writes a DTEL++ policy to describe access rights, controlling both the access of users to the application and access of application components to each other. The DTEL++ policy refers to methods declared in CORBA IDL, and in this way, it is like QuO's other specification languages. The DTEL++ policy must cover all of the interfaces used in the application, including those used by QuO callbacks and by alternate behavior specifications.

The OO-DTE access control code is currently inserted into a QuO application using CORBA interceptors in each ORB. Thus, when a client invokes a method on an object, the OO-DTE access rights are checked by an ORB interceptor after the QuO client's delegate has processed the invocation. This fact about sequencing can be important when the delegate makes a callback or doesn't pass the invocation through directly when implementing adaptive behavior: the access rights for the original invocation are checked when the invocation is passed through the delegate to be handled by the ORB.

The application developer must also define a set of cryptographic certificates needed by the application. OO-DTE does not assume that an application's components are all trusted to the same degree. Instead, each client and object must authenticate each other using cryptographic means (e.g., SSL [Net]).

5.2.3. Protecting the QuO Infrastructure

Just as application clients use CORBA to invoke methods on application objects, so QuO delegates use CORBA to invoke methods for accessing system condition objects, for initializing contracts, and for causing the kernel to evaluate contracts. The QuO infrastructure is therefore subject to the same kinds of attack as is the application. For example, a malicious program could try to trigger QuO's adaptation mechanisms at the wrong time by changing the value of a system condition, or to disable QuO altogether by changing QoS contracts.

We protect the QuO infrastructure using OO-DTE access control just as for applications. The QuO infrastructure code must use ORBs with OO-DTE interceptors (in fact, if this were not so, OO-DTE would not work because the infrastructure and the delegates could not establish mutual authentication) and the DTEL++ policy must describe access control for infrastructure methods also. The policy must prohibit delegates from damaging the infrastructure but still give them the access they need for adaptation.

The need to protect the QuO infrastructure has implications for the design of QuO. We assume that threats to QuO come from application software and not from code within the QuO infrastructure we supply. Then the QuO design must not allow application software to run in the same process address space as the QuO infrastructure. OO-DTE cannot protect QuO from malicious code in the same address space because that code may bypass CORBA altogether and directly access QuO code and data structures. The infrastructure we supply with the QuO distribution is therefore defined to be part of the trusted computing base (TCB) [DoD85] and all extensions to that infrastructure and applications must run in other address spaces. For example, the QuO kernel cannot be run securely in the same process as an application client, even though doing so is one way to obtain enhanced performance. These sorts of property tradeoffs are fundamental issues in distributed computing technology.

5.3. Building Survivable Applications Using the QuO Toolkit

In this section, we describe two of the demonstration applications we have developed to test and evaluate the flexibility and utility of the QuO toolkit in supporting more survivable, adaptive applications.

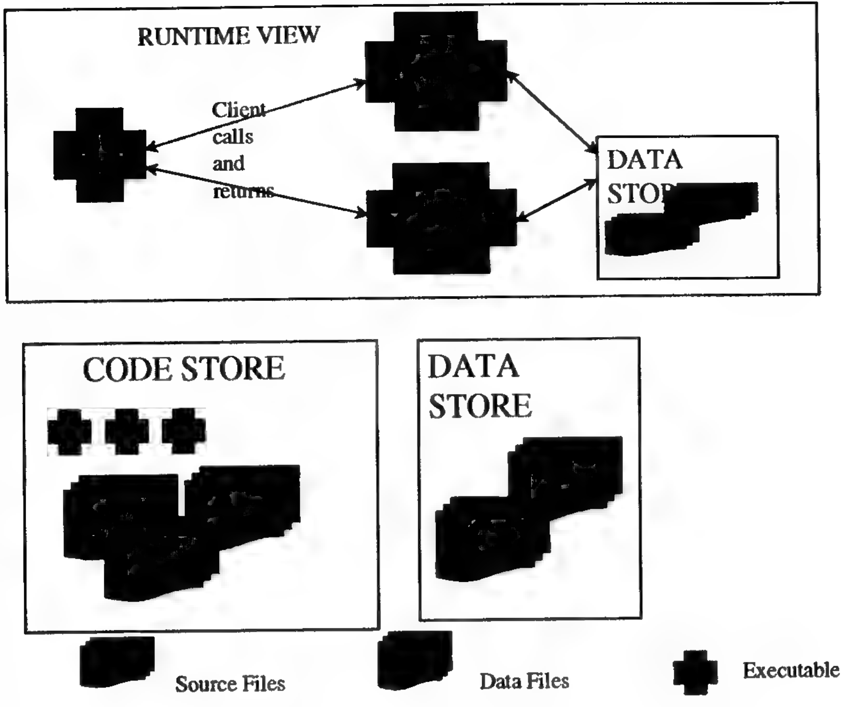

Figure 8: The survivable inventory application

5.3.1. Example Application Integrating IDSs with QuO Adaptivity

In this section, we describe an application demonstrating how a commercial IDS, a simple custom developed IDS, and application-specified intrusion detection are all integrated to provide intrusion awareness and adaptive behavior in response to intrusion detection at the application level. This application is fairly simple, but illustrates a number of important capabilities, including the following:

- Integration of commercial and non-commercial IDSs using the QuO framework.
- An application seamlessly interfacing to multiple IDSs, enabling the IDSs to cooperate through the application layer and increasing intrusion coverage.
- An application participating in the intrusion detection process, by recognizing conditions that can indicate intrusions but that are not detected by the IDSs.
- An application adapting to survive potential intrusions, triggered by outputs of the IDSs.

In contrast, the application can use the IDSs as mechanisms to turn on, turn off, or change the level of intrusion detection provided based upon its operating mode and security needs.

5.3.1.1 Overview of the Experimental Survivable Application

This experimental application implements an inventory application with a fixed set of inventory items, as illustrated in Figure 8. Inventory data is stored as files in a designated data directory. Two servers manage the inventory: one more secure than the other. The client program, representing the inventory control system, provides a user interface through which users can identify themselves (i.e., log in), add or consume items in the inventory, and log out. Both servers can respond to requests from the clients, but the more secure one authenticates (using a simple authentication scheme) each request and grants access only to certain clients.

This is an example of the alternative behaviors that the QuO middleware is intended to mediate. In normal mode, all client requests are serviced by the non-authenticating (and therefore faster) server. As conditions indicate that intrusions are more likely, the inventory control system adapts to use the authenticating server and then, eventually, may cut off all access to non-privileged users. The client and server programs are simple CORBA objects. No intrusion detection or adaptation is programmed into them. For this example, all adaptation is built into the QuO middleware layer. We utilize three intrusion detection instruments in this example:

- Tripwire, a commercial file system integrity checker [Kim94];
- FileCounter, a simple, custom developed directory access checker; and
- Specifications, encoded into the QuO contract regions, indicating the expected round trip response time range and recognizing when client requests are being abnormally delayed (possibly because they are being intercepted, or because of host or network attacks). Note that such a delay, by itself, is not sufficient indication of an intrusion: benign network congestion could cause a false positive. This merely serves as an example of an indicator of potential problems for an intrusion aware application.

We use Tripwire to monitor the file system that stores the source and executable code of this application and FileCounter to monitor the data directory. We use system condition objects to interface to Tripwire and FileCounter, each of which normally provides its own custom interface. These system condition objects project values from the IDSs to the QuO layer and provide common access to the control interfaces provided by the IDSs. Tripwire's system condition object (called IDSValue) projects a value to the QuO contract indicating whether the integrity of the code store has been violated (in Tripwire's view). Similarly, FileCounter's system condition object (called FileAddedOrDeleted) projects a value indicating to the QuO contract indicating whether a data file has been lost or added.

Deviation from normal operating behavior often points toward potential problems. For instance, if a server returns a value that does not make any sense in the current context, the client may become suspicious that the server has been compromised. Similarly, if it takes an abnormal amount of time to fulfill a request to the inventory server, the client may become suspicious that there is a problem in the network, a host, or in the server. It is straightforward to encode such normal operating ranges in QuO's contract regions and to specify adaptive behavior to trigger when the application falls outside normal ranges. In this example, we use the contract and a simple system condition object (called TimeTaken) to measure the average round trip time of method calls and watch whether it falls outside of the expected range. As we stated earlier, there may be a variety of causes of this abnormal behavior, only some of which are the result of intrusions. Determining the actual cause falls somewhere between system troubleshooting and intrusion detection.

5.3.1.2 Basic Integration Architecture

The example application includes the three system condition objects described above: IDSValue, FileAddedOrDeleted, and TimeTaken. IDSValue, illustrated in Figure 9, provides the QuO interface to Tripwire. Tripwire can be initialized to monitor specific sections of a file system for particular attributes, such as permissions or modification times, of the files and directories in that section. Tripwire computes a database upon initialization. At runtime, it recom-

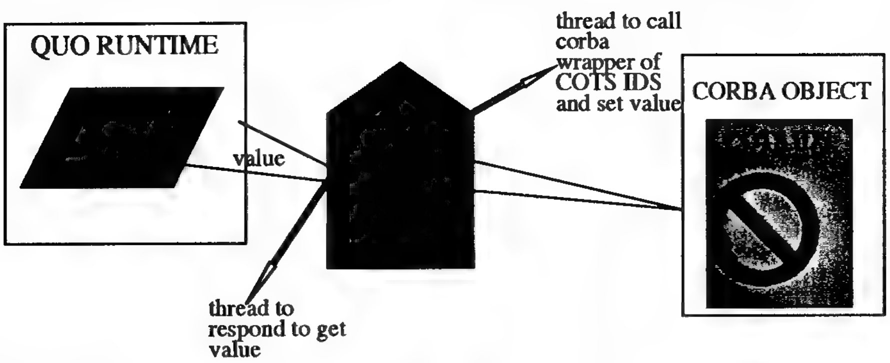
QUO RUNTIME
value
thread to call corba wrapper of COTS IDS and set value
CORBA OBJECT
thread to respond to get value

Figure 9: A system condition object interface to a COTS IDS

putes the database, compares the newly computed database against the initial one, and presents a result set that indicates what occurred in the file system section between the runs.

We wrap Tripwire with a CORBA object interface that runs Tripwire periodically and analyzes its output. If there is any change in the file system section this CORBA object returns a value 1, otherwise it projects a value 0. The IDSValue system condition object is hooked to this CORBA object. One of its threads polls Tripwire's CORBA wrapper to get the latest value. The other thread responds to requests from QuO contracts for the latest value. Similarly, FileCounter's system condition object, FileAddedOrDeleted, projects a value indicating to the QuO contract indicating whether a data file has been lost or added.

Of course, if the IDS system were a CORBA object already, then no CORBA wrapper is necessary. The other IDS component is a CORBA object that monitors files in a directory. FileCounter produces a value 1 if a file is added or deleted in that directory, 0 otherwise. We have used this as a simple custom developed IDS and integrated it with QuO along with Tripwire, in order to experiment with multiple IDS inputs.

The QuO contract used in this example defines the following operating regions, each defined in terms of the system condition objects described above:

- NORMAL : (TimeTaken < 500 ms) and (IDSValue = 0) and (FileAddedOrDeleted = 0)
- TIME SUSPECT : (TimeTaken >= 500 ms) and (IDSValueSC = 0) and (FileAddedOrDeleted = 0)
- ACCESS SUSPECT : (IDSValueSC = 1) xor (FileAddedOrDeleted = 1)
- INTRUSION LIKELY : (IDSValueSC = 1) and (FileAddedOrDeleted = 1)

When a client is activated, it obtains a QuO delegate instead of a CORBA stub. All remote invocations initiated by the client are passed through the delegate. The delegate has associated with it the mechanism to determine the current contract region. Depending on the current region, the delegate makes different dispatching decisions about the remote invocation, which leads to the application's adaptive behavior. In this demonstration, the delegate is programmed to cause the following adaptive behavior based on the current contract region:

- NORMAL region: client requests are forwarded to the non-authenticating server.

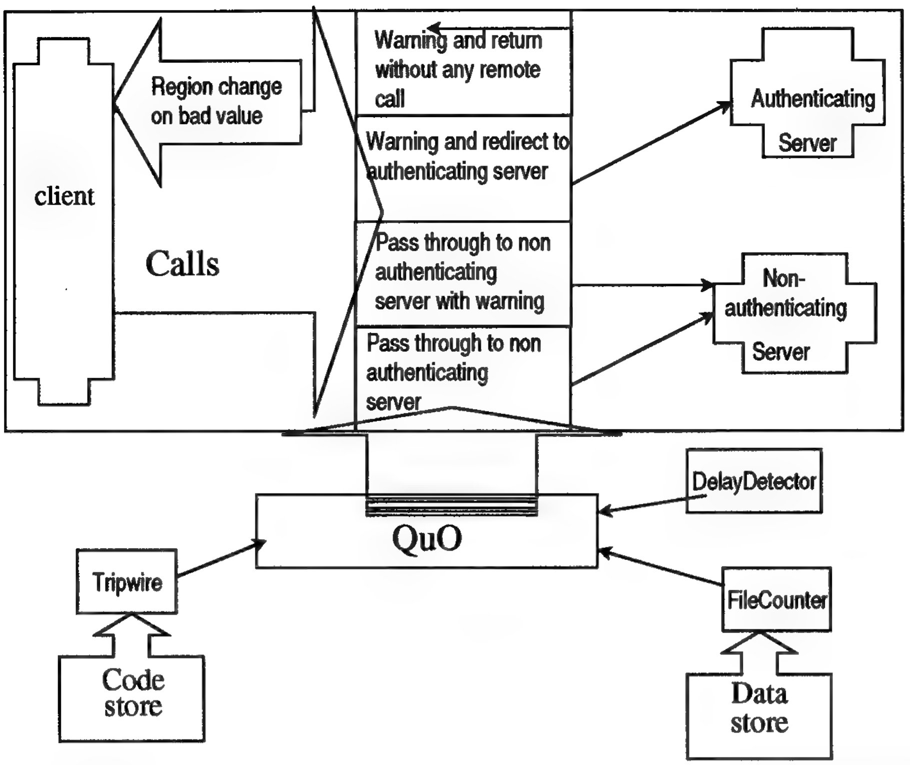
client
Region change on bad value
Calls
Warning and return without any remote call
Warning and redirect to authenticating server
Pass through to non authenticating server with warning
Pass through to non authenticating server
Authenticating Server
Non-authenticating Server
DelayDetector
QuO
Tripwire
FileCounter
Code store
Data store

Figure 10: Runtime behavior of survivable inventory application

- TIME SUSPECT region: A warning message is displayed to the user notifying of the unusual delay and urging caution in using the inventory system. The client's requests are still forwarded to the non-authenticating server.
- ACCESS SUSPECT region: A warning message is displayed to the user stating that a potential access violation is detected and that requests may or may not be granted. Clients' requests are now forwarded to the authenticating server, which grants access only to privileged users.
- INTRUSION LIKELY region: A warning message is displayed stating that it is highly likely that there was an intrusion that could have compromised the code and data store of the application, and all client requests are returned without making any remote call. This implies that only some inventory operations (i.e. that could be handled locally, for example a query about an inventory item could be answered, with some degree of accuracy, based on the value last seen) are available at this region. One can extend the range of available operations by using various technologies such as object caching or maintaining a local replica of the inventory server.

In addition, if at any time the return value is negative (under normal circumstances, the server should never return a negative value) that value is reported to the user and the contract region is switched to ACCESS SUSPECT. All of these adaptive behaviors are specified in QuO's specification languages. Figure 10 presents a pictorial representation of the full IDS aware inventory application after the QuO-IDS integration.

5.4. *Using Dependability for Survival*

There are numerous tools and mechanisms, which we generically refer to as property managers, that manage low level system resources. These property managers provide the capabilities that QuO applications must measure and control in order to achieve and adapt to levels of service in the system. For example, we have developed a bandwidth management system that uses RSVP to reserve network bandwidth, providing improved network response for the application [BBN98]. Similarly, we have built an availability example around the Proteus dependability management system [Cuk98], which uses group communication, replication, and fault recovery to provide higher levels of availability to the application. These integrated property managers can be used to develop adaptable, survivable applications in the following ways:

- They can provide information indicating anomalous behavior and its causes. In general, more precise information means improved adaptive response. For example, unusual delay (which we have used as an indicator of a potential problem) could be caused by the network, by a compromised object, a crashed object, a compromised host or a crashed host. Given additional information, the application can adapt intelligently.
- They can provide the application more adaptation opportunities. For example, if it is the network that is the source of an abnormal delay, the QuO application can attempt to reserve bandwidth, if such a manager is available.

Dependability is one such property. Under this project, we enhanced the capabilities of the Proteus dependability manager [Sab99] and integrated it into an example in order to illustrate how other property managers working in concert with IDSs and adaptable applications can produce more flexible, survivable applications.

The QuO-IDS integration example as described in the previous section, although survivable in the face of some types of intrusions, is hardly dependable. If one of the server objects dies, the whole application dies. Using the Proteus dependability manager, it is possible to make the application more dependable in the sense that it can tolerate a certain number and type of faults. Proteus achieves that by replicating the server objects on multiple hosts. However, in the course of its fault recovery, Proteus usually hides faults from the application. That is, when an object crashes, Proteus restarts it and updates its state, to maintain a level of dependability transparent to the application. In the context of a survivable, intrusion-aware application, fault masking may hide potential clues for intrusion.

In conjunction with our research, the University of Illinois has developed a fault notification interface for Proteus. Using this interface we have developed a CORBA object, called FaultObserver, that receives notification from Proteus about faults such as the unsuccessful start of a replica, crash of a replica, and crash of a host. Each notification consists of a set of fault information that can be stored and analyzed to recognize patterns of failures that might indicate an intrusion. The following are two example conditions that FaultObserver currently recognizes:

- POTENTIAL INTRUSION OF HOST *x*: this indicates that either the host named *x* crashed or replica start attempts on this host were unsuccessful.
- POTENTIAL COMPROMISE OF OBJECT *o*: this indicates that either a replica of object *o* has crashed or attempts to start a replica of object *o* have been unsuccessful.

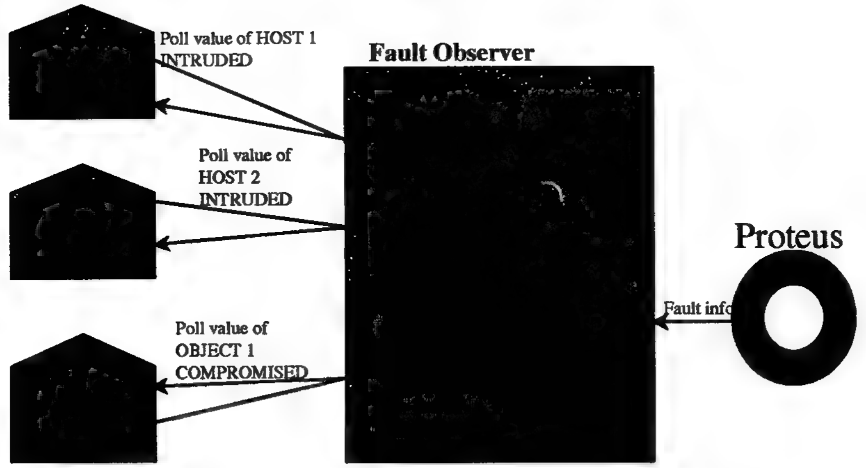
Poll value of HOST 1 INTRUDED
Fault Observer
Poll value of HOST 2 INTRUDED
Proteus
Fault inf
Poll value of OBJECT 1 COMPROMISED

Figure 11: Using dependability manager to support intrusion detection and response

Figure 11 shows three system condition objects that we have hooked up to the FaultObserver object. Each of the top two projects the value of the POTENTIAL INTRUSION conditions for one of the two replication hosts and the bottom one projects the value of the POTENTIAL COMPROMISE condition for a server object.

Let us consider a simple client-server application that uses Proteus to replicate the server and in addition, also maintains a non-replicated server. Replication provides the fault tolerance and dependability, whereas the non-replicated server makes it possible for the application to bypass the replication mechanism if it chooses to. A contract for this application may include the following regions predicated on the system condition objects described above:

- HOST SUSPECT: the HOST INTRUDED condition is true for one or both replication hosts.
- SERVER SUSPECT: the OBJECT COMPROMISED condition is true for the non-authenticating server.

The application's adaptive behavior may include:

- If the application is in the SERVER SUSPECT region, the client's requests will be redirected to the different non-replicated server object.
- If the application is in HOST SUSPECT region, Proteus will be asked not to place replicas in the intruded host(s).

In addition to the notification of crash faults, which Proteus already provided, the University of Illinois has enhanced Proteus under this project to provide notification for some value faults that could prove useful for a survivable application.

Apart from the obvious benefit of replicating the inventory servers, the *redundancy* and *monitoring* as described above may lead to further *adaptive* behavior that improves the survivability of the inventory application in future examples. For instance, when Proteus indicates that it cannot maintain the desired number of replicas under the existing circumstances, the application can add a new replication host. If the monitoring mechanism (via a system condition object) indicates that a particular host is potentially compromised, that host could be removed from the list of replication hosts, thereby removing replicas from that host. Similarly, if the monitoring mechanism indicates the non-authenticating server is potentially compromised then the client's request will be redirected to the authenticating server.

Future versions of Proteus can also manage and exploit diversity at lower levels to the advantage of the application that aims to increase its survivability. For instance, replicated objects could be on different kinds of hosts such as Linux, Solaris and NT. Given this, the application could *adapt* to use only Solaris hosts if there is an NT virus alert. As another example, Proteus provides a choice of replication schemes (ranging from active to passive replication) and various voting mechanisms. Given that, if too many of the replication hosts are down, the application can switch from active to a more passive replication scheme and yet still provide some degree of dependability and fault tolerance.

6. Lessons learned

In the course of this research, we developed some experience about the nature of middleware adaptation, application assisted intrusion detection, and the feasibility of the approach to survivability that integrates together a number of more localized protection and security mechanisms to achieve more effective coverage. In this section we discuss some of these areas: middleware supported adaptation, using multiple complementary IDSs, integrating off the shelf IDSs, integrating security property management, and application strategies that can complement infrastructure based detection and protection mechanisms.

6.1. Middleware Supported Adaptation

Through this work we have demonstrated that it is possible to develop applications that can measure and adapt to the changes in their environment. Furthermore, we have demonstrated that such applications can be developed using a common set of *middleware abstractions*, similar to those that have simplified the development of distributed applications. This is in contrast to the ad hoc manner in which adaptive, QoS aware applications are currently built.

The common set of abstractions that we have developed center around the need for an application to *specify*, *measure*, *control*, and *adapt to changes in* its environment. Thus we provide languages, abstractions, and runtime support to support *contracts*, which divide system properties into regions of concern; *system condition objects*, which provide interfaces to measure and control system properties; and *delegates* and *transitions*, which trigger adaptation in the face of changing conditions. We learned early in this project that without such abstractions, the development of applications falls into two categories:

- The development of applications using high-level programming paradigms, such as distributed object computing, but that are brittle, don't scale well, and have little option but to fail when faced with unpredictable environments.
- The development of robust, scalable, and adaptive applications, but in an ad hoc, non-reusable manner. If DOC programming standards are used at all, they must be programmed around in order to access the non-functional systemic capabilities, resulting in code that is difficult to maintain.

We have demonstrated that the abstractions and capabilities that we have produced support many dimensions of systemic, or QoS properties, including real-time, fault tolerance, dependability, managed communication, security, and survivability. Supporting these abstractions in middleware has proven to have the following advantages:

- It supports the addition of QoS concerns to existing applications
- It supports the separation of the roles of *application developer*, who needs in-depth knowledge of the functional requirements of the application, from that of *QoS developer*, who needs in-depth knowledge of the possible environmental and system conditions.
- It advances the concepts of *aspect-oriented programming* [Kic96a] and *open implementation* [Kic96b], which advocate the separation of functional and QoS concerns.

Furthermore, time and continued research has demonstrated that at least some of the abstractions and capabilities that we have developed are being recognized as natural extensions to the DOC standard. When we began this work, no support existed in CORBA for the abstractions we developed (i.e., delegates, contracts, system condition objects, and gateways). Recent additions to the CORBA standard have incorporated some of the abstractions so that future versions of QuO will extend CORBA with fewer components, instead using features of the standard to achieve the same functionality. For example, the functionality of the delegate, which was not supported at all when we designed it, is now supported by CORBA's *smart proxies* and, to a limited degree, by CORBA interceptors. Likewise, many of the functions of our original QuO gateway are now supported by CORBA's *pluggable protocols*, in which our newest version of the QuO gateway is implemented.

Further research will show whether the abstractions that we've developed are a complete set. We've already encountered a need for dimension-specific QoS languages, such as security policy languages like DTEL++ or real-time languages like TAO's RIDL. These are, so far, orthogonal to our QDL languages.

6.2. Multiple, Complementary IDSs and Managers

It has been shown from an experiment conducted by MIT Lincoln Laboratory for DARPA that multiple IDS systems working together can be more effective in identifying real attacks. MIT LL evaluated a number of IDSs, testing them on a number of different types of attacks, and scoring each according to the number of attacks that it detected and the number of false alarms it raised.

The results indicated that while none of the IDSs overwhelmingly detected most of the attacks, the hypothetical combination of the best detectors for each attack resulted in more than two orders of magnitude reduction in false alarm rates while improving detection accuracy over commercial and Government keyword-based systems [Dyn98]. This provides the motivation for a model where there are multiple IDSs operating concurrently, and the need to organize and integrate their operation to achieve intended application oriented improvements in survivability.

One of the strengths of the QuO framework is that it provides simplified support for interfacing to multiple managers and mechanisms. Part of our current development and experimentation involves integrating multiple managers and mechanisms. The first example we developed, described in Section 5.3.1, combined a commercial off the shelf IDS with a custom developed (but simplified) one tailored for the specific need. The second, described in Section 5.3.2, uses the Proteus dependability manager in combination with the Tripwire IDS. This has two potential application survivability benefits. First, fault detection information collected routinely as part of dependability support can be used to aid the intrusion detection system, and second, the reconfigurability of replication assets can be used to help recover from detected intrusions.

The examples that we have developed, despite having limited interaction between managers, have delivered the expected benefits. Some managers, such as multiple IDSs, security policy managers, and dependability managers are complementary and can produce higher levels of service when used together.

6.3. Integrating COTS IDSs

If the concept of using multiple special purpose IDS systems in concert is to be viable and extensible, then we need to be able to take off the shelf IDS systems and easily insert them into various application contexts. To test our approach to this type of integration, we used the example described in Section 5.3.1. The collection of intrusion detection systems available to us was limited, largely to those sufficiently mature under development as part of the DARPA Information Survivability program, and those inexpensively available commercially. From these we chose two to work with: Tripwire, a commercially available ID discussed earlier (and successfully integrated with our concept example), and JAM, an experimental ID under development within the DARPA IS program.

JAM [Sto97] is essentially a classifier system that employs learning and meta-learning techniques to build and refine the classifier. JAM has been used successfully to learn intrusion patterns in system traces [Dyn98].

One of the major problems we encountered in trying to integrate JAM with QuO is a mismatch of modes of operation. JAM currently operates in a batch mode. Although it is possible to ask it to classify a data set in an interactive manner, the current version does not provide any easy way to do it. Off-line usage provides only a small experimental footprint to complement the current runtime adaptation in QuO, either as a source of inputs for contract evaluation or as a mechanism to provide its service. Because JAM is geared towards stand alone usage with a GUI and not embedded usage as envisioned in integrating into a QuO environment, it did not have an appropriate API which could easily be used to integrate into QuO's system condition constructs. Additionally, it turns out to be a complex job to create the data definition and data sets that JAM would need in the context of and training for a new application such as integrating with QuO. Because of these issues, the experiment to integrate JAM as one of the ID systems in our adaptive application context has been postponed, pending the addition of online usage interfaces to JAM.

In general, the issues we encountered in integrating with Tripwire and in trying to use JAM fall into two broad categories:

- Interface Requirements: How does an application interface with a COTS IDS? The best possibility would be a runtime service provided by the COTS IDS. A programmable API would be the next best choice. The minimal requirement is that it should be possible to run the IDS from a CORBA object and communicate results in and out. A runtime service or a programmable APIs make this task easier, but human oriented interfaces have been encapsulated successfully as well, most often with less flexibility.
- Integration Architecture: What is an appropriate level of integration? We think that the way we have architected the integration by means of a CORBA wrapper that acts as a peer of a QuO component is a general pattern of usage that will be repeated with other COTS systems. Depending on the layer in which the QuO component operates, the COTS system

may provide inputs to contract evaluation or may provide some service. If the COTS system is already a CORBA object itself, no CORBA wrapper will be needed.

6.4. *Integrating Security Property Management*

Based on our experience thus far with using OO-DTE in QuO, we can make several observations. First, we learned that incorporating OO-DTE access control into QuO was a straightforward development task. Because OO-DTE is implemented as CORBA interceptors, integrating it required only minimal changes to QuO code. Using OO-DTE successfully was largely a matter of setting up the policy and cryptographic certificates correctly for each application. Second, we expect that using OO-DTE will not be as straightforward in the presence of mechanisms that support other properties. For example:

- Although access control in the presence of fault tolerant replication is conceptually simple (just give each replica the same access rights), the actual implementation appears harder. In addition to handling application-level invocations, the replicas must run some replica coordination protocol. It is not yet clear what access rights are required in this protocol.
- Using access control in the presence of a realtime ORB [Sch98] will mean porting the access control mechanisms to an alternative ORB and ensuring interoperability between ORBs.

Third, building security-aware QuO applications will mean allowing applications to have direct access to the security policy to inspect it and under certain circumstances with the appropriate permissions possibly to modify it. Currently this access is not possible. To make it possible we must encapsulate the OO-DTE policy in a CORBA object and define access methods for the policy. Once that is done, the access rights themselves must be access controlled according to some meta-level security policy.

6.5. *Applications Participating in Intrusion Detection*

Our work with the QuO framework, various property managers, and the integration of ID and security has shown that many of the QoS properties with which an application is concerned are the same QoS properties that can indicate an intrusion or attack. For example, by definition denial of service attacks will manifest themselves by an application losing some service upon which it is dependent. Likewise, flooding attacks, attacks on particular hosts, networks, or processes can manifest themselves as changes in the system conditions monitored by QuO applications.

We illustrated in our example application described in Section 5.3.1 that QuO applications can specify normal and abnormal patterns of behaviors in their regions and recognize when the system is operating outside these regions. Many of these, especially if coded carefully, can indicate patterns of attack. Slow service, loss of network resources, abnormal responses by server objects, etc. can all indicate potential attacks. The application can aid IDSs by alerting them toward potential intrusions that should be analyzed or by indicating conditions in which multiple IDSs should be deployed.

In addition, QuO's system condition objects and instrumentation normally used for bottleneck identification and to drive resource management decisions, can also be used to collect system information over time that might recognize intrusions that are difficult to recognize from small

```
contract <contract_name> (
  syscond [ nowatch ] <idl_type> <impl_type> <sc_name>,
  callback <idl_type> <cb_name>,
  ... )
{
  [ static ] syscond [ nowatch ] [ probe ] <impl_type> <name> (<arg> , ... );
  ...

  region <name> ( <predicate> ) {
    region <name> ( <predicate> ) { ... }
    transition <name> -> <name> { ... }
    ...
    precedences <region_name> ...;
  }
  ...
  transition any -> <name> { ... }
  transition <name> -> any { ... }
  transition inactive -> <name> { ... }
  transition <name> -> inactive { ... }
  ...
  precedences <region_name> ...;
};
```

Figure 12: Representative CDL syntax

windows or groups of events, such as slow degradations in service. This information can feed into off-line analysis capability, such as that currently provided by JAM. One weakness of anomaly detection IDSs is that they can be trained by intruders over time to recognize anomalous behavior as normal. An application could aid in detecting these types of intrusions by collecting information indicating slow, deliberate degradations of service or changes in behavior patterns.

Finally, we have also shown, in the example application described in Section 5.3.2, that other property managers and mechanisms can also be useful in intrusion detection. We have concentrated our initial efforts on using the Proteus dependability manager, which in normal usage would attempt to mask faults that could indicate intrusions, to collect information about fault patterns, as a means of helping recognize intrusions. However, other mechanisms, such as resource management, realtime scheduling, and instrumentation, could also be focused on the job of intrusion detection.

7. Syntax and Semantics of the Quality Description Languages

7.1. CDL Syntax and Semantics

Figure 12 illustrates the representative syntax of a CDL contract. The contract definition starts with the keyword **contract**, followed by the name of the contract class and the list of contract parameters. The contract parameters are system condition objects and callback objects that are created outside the contract and passed in (e.g., because they are shared with other contract instances, the client, or other objects in the system). The contract parameters are one of the following forms:

syscond [nowatch] <idl_type> <impl_type> <name>
callback <idl_type> <name>

The keyword **syscond** or **callback** indicates whether the parameter is a system condition object or a callback object; <idl_type> is the IDL class of the object; and <name> is the name by which the object is referenced in the contract. If the parameter is a system condition object, the name of the implementation class must also be specified (<impl_type>). System condition objects can also be observed (i.e., a change in the system condition object triggers contract evaluation) or not (i.e., changes in the system condition object go unnoticed until the next contract evaluation). The default is for each system condition object to be observed. The optional keyword **nowatch** tags the system condition object as non-observed.

Next is a list of system condition object declarations local to the contract. Each of these is identified by the keyword **syscond**. The keyword **static** preceding **syscond** is optional; the default is non-static. Non-static system condition objects are local to each instance of a contract class, i.e., a new one is created each time an instance of the contract is instantiated. Static system condition objects are shared by all instances of a single contract class; one is created when the first instance of the class is instantiated. The other parts of the system condition object declaration are the implementation class of the object (<impl_type>), the local name of the object (<name>), and a list of actual parameters passed to the object constructor (<arg> ...). The IDL type is not necessary because the object will be local to the contract class; i.e., it will manipulate it as a Java object, not a CORBA object.

The next section of the contract definition contains declarations of regions, transitions, and precedences. Each region declaration can have declarations of regions, transitions, and precedences nested within it. There can be any number of levels of nested regions. Each region is declared by the keyword **region**, followed by a name, a predicate statement in parentheses ("(...)"), and a body in brackets ("{...}"). The body of a region can contain nested regions or it can be empty.

The **region** declarations, **transition** declarations, and **precedence** declaration can be in any order. However, the declarations for regions referenced in transition or precedence statements should appear before they are used. Every contract will have at most one region at any level that is active at a time. However, region predicates can overlap and it is possible for the predicates of more than one region to be true at any given time. The **precedences** statement is a way for the programmer to specify which regions should be active if more than one has a predicate that is true. The **precedences** statement is optional; the default precedence is the order in which the regions are listed.

Transitions specify behavior that is invoked when the active region changes. A **transition** need not be specified for every possible region change and no transitions need be specified for any level of region. A transition is specified using the keyword **transition** followed by two region names separated by the operator "**->**", indicating a transition from the first region to the second region. One of the region names, but not both, can be replaced with one of the keywords **any** or **inactive**. The **transition** declarations are interpreted as follows:

- **transition** <region1> **->** <region2> is triggered when the contract, with active region <region1>, is evaluated and <region2> becomes the new active region.
- **transition any ->** <region1> is triggered when the contract, with an active region that is not <region1>, is evaluated and <region1> becomes the new active region. It doesn't matter what region was active previously as long as it was not <region1>.
- **transition** <region1> **-> any** is triggered when the contract, with active region <region1>, is evaluated and another region, not <region1>, becomes active.

- **transition inactive ->** <region1> is triggered when the contract has no active region and it is evaluated with <region1> becoming active.
- **transition** <region1> **-> inactive** is triggered when the contract, with active region <region1>, is evaluated and no region becomes active, i.e., none of the region conditions is true.

Each transition must have at least one region name in its declaration, i.e., the following have no meaning:

- **any -> any**
- **inactive -> any**
- **any -> inactive**
- **inactive -> inactive**

The body of a transition, delineated by brackets ("{...}") contains a list of methods to be invoked when the transition is triggered. Each of these must be a method on a callback or a system condition object. The methods are separated into **synchronous** and **asynchronous** sets. Either synchronous or asynchronous behavior can be specified, or both. The contract will wait for synchronous callbacks to finish before it can be evaluated again. Asynchronous callbacks are spawned. Therefore, transition statements can have any of the following forms:

```
transition <from_region> -> <to_region> {
  synchronous { <method_call> ... }
  asynchronous { <method_call> ... }
}

transition <from_region> -> <to_region> {
  asynchronous { <method_call> ... }
  synchronous { <method_call> ... }
}

transition <from_region> -> <to_region> {
  synchronous { <method_call> ... }
}

transition <from_region> -> <to_region> {
  asynchronous { <method_call> ... }
}
```

The method calls in transition behavior can be methods on callback objects and system condition objects declared previously in the contract. Arguments can be passed to these methods, including constants of CORBA primitive types (e.g., long, char, string, double) or variables referring to system condition objects. A variable referring to a system condition object can be passed as either an object, simply by using its name, or as the value held by the system condition object, by specifying a CORBA simple type for it in parentheses before the variable name. For example, the following transition behavior:

```
ReplMgr.adjust_degree_of_replication(ClientExpectedReplicas);
```

passes `ClientExpectedReplicas` as an object, while the following:

```
ReplMgr.adjust_degree_of_replication((long) ClientExpectedReplicas);
```

passes the value of `ClientExpectedReplicas` as a long.

7.2. SDL Syntax and Semantics

The basic structure of an SDL description is as follows:

```
delegate behavior for interface interface_name and contracts contract_list is
  delegate_variable_declarations
  adaptive_behavior_descriptions
  default_behavior_description
end delegate behavior;
```

The *interface_name* must be an interface defined in an IDL file being parsed on the same quogen run. The interface name can be fully qualified, i.e., if the interface *I* is defined within a module *M* then `interface_name` will be *M::I*. See the *QuO Connector Setup Language (CSL) Reference Guide* for further details on how to parse IDL files in a quogen run and how to instruct quogen to generate delegates for an interface.

The *contract_list* must be of the form *c1, c2,* ... where each *ck* is the name of a contract defined in a CDL file being parsed on the same quogen run. See the *QuO Connector Setup Language (CSL) Reference Guide* for further details on how to include CDL files to be parsed in a quogen run.

The *delegate_variable_declarations* section allows the programmer (i.e., the QoS developer) to specify variables that are initialized during the delegate creation and initialization time. These become member variables of the delegate class. Each *delegate_variable_declaration* specifies a local name to which it can be referred from within anywhere in the SDL file (i.e., the scope of these variables are the whole SDL file).

The *adaptive_behavior_descriptions* section contains a list of method calls and/or returns for which the delegate defines alternative behaviors. Each of these in turn specifies a list of contract regions and behavior descriptions that, if the contract region is active when the method is called and/or returned from, will cause the delegate to execute the corresponding behavior.

The *default_behavior_description* allows the specification of behavior to be executed when any method that is not specified in the SDL description is called.

7.2.1. Delegate Variable Declaration

Delegate variables have the following basic syntax :

<type name> *<variable name>* [*<bind clause>*] ;

where *<type name>* is either an IDL style fully qualified name (of the form *module::type*) or a simple name (e.g., long). The SDL code fragment in Figure 13 shows five variations (labeled by

```
Delegate behavior for interface InvInstrumented
                  and contracts InstrumentedContract is
 Inv altRemoteObject1 bind with name InventoryServer;     //example 1
 Inv altRemoteObject2 bind;                               //example 2
 long my_count;                                           //example 3
 Instrumentation inst bind with java_code {               //example 4
        new StandardInstrumentation();},
                          with cplusplus_code {
                                   new StandardInstrumentation();};
 string my_string bind with java_code{inst.genUID();},    //example 5
                          with cplusplus_code{inst->genUID();};
```

Figure 13: Possible declarations of local variables inside a delegate.

numbers 1 –5 in comments) of delegate variable declaration. These declarations generate protected member variables of the delegate class.

The *<type_name>* in a delegate variable declaration can be one of the following:

- *Basic CORBA IDL type* – These map to the proper Java and C++ types according to the CORBA language mapping.
- *IDL-defined types* – These are mapped to the proper Java or C++ type as defined by the respective language mapping. For Java, this means that it will figure out the proper package name which could be influenced by the –p and –idl2package flags used at the code generator's command line.
- *QuO pseudo type* – QuO pseudo types are types that are not defined in the quo IDL module but are assumed to exist (similar to CORBA pseudo types). Currently quo::COSHelper and quo::Connector are two quo pseudo interfaces. For Java these map to `com.bbn.quo.[COSHelper|Connector]`. For C++, these map to `COSHelper*` and `Connector*`.
- *Everything else* - *<type name>* is dumped as a Java type (i.e. `m::n::foo` as `m.n.foo`) or a C++ pointer type (`m::n::foo` as `m::n::foo*`).

If you specify a bind clause in a variable declaration (examples 1, 2, 4 and 5 in Figure 13), it produces two public member functions in the delegate class. One is an initializer method and the other is a setter method for the corresponding protected member variable. For instance the second declaration in Figure 13 produces the following two methods in C++:

```
void initialize_altRemoteObject2();
void set_altRemoteObject2(Inv_ptr arg1);
```

assuming that Inv is defined as an interface in an IDL file that is parsed during the quogen run that produces the delegate (see the *QuO Connector Setup Language (CSL) Reference Guide* for how this is done). In Java these two look like:

```
public void initialize_altRemoteObject2();
public voidset_altRemoteObject2(com.bbn.quo.examples.idsinv.gen.Inv arg1);
public void initialize_my_string() {
      my_string = inst.genUID();
}
```

(a) Java

```
void initialize_my_string() {
      my_string = inst->genUID();
}
```

(b) C++

Figure 14: Code generated from example 5 in Figure 13

assuming that the command line flags to quogen are set to instruct the **idl2java** compiler to generate the Inv class in the `com.bbn.quo.examples.idsinv.gen` package.

The setter methods simply take an argument of the appropriate type and set the corresponding variable. One can use the setter methods to set the variable to point to an already bound object.

The initializer methods do what the bind clause in the corresponding declaration specifies. Currently the following types of bind options are supported:

- **Simple bind** (example 2 in Figure 13) uses Visibroker's *bind* function to obtain a reference to a remote object stub. Visibroker provides a bind() method in the classes that it generates from IDL interfaces. QuO 2.1 uses Visibroker for Java and TAO for C++. If you try to use a simple bind declaration to generate a C++ delegate, you will get an SDL warning stating that bind is not supported for C++ (if you have configured your CSL file to print warning messages. See the *QuO Connector Setup Language (CSL) Reference Guide* for how to do this).
- **Bind with name** (example 1 in Figure 13) uses quo::COSHelper to obtain a reference by resolving the string name specified in the declaration.
- **Bind with verbatim** (examples 4 and 5 in Figure 13) uses the Java and/or C++ code specified in the declaration verbatim to initialize the corresponding variable. For instance, declaration (5) will generate the initializer methods in Figure 14.

7.2.2. Adaptive Behavior Descriptions

The *`adaptive_behavior_descriptions`* section consists of a (possibly empty) list of adaptive behavior descriptions, each keyed on a method call or return. Basically, each says, "when the client calls (or returns from) this remote method, look at the current state of QoS (as indicated by the active regions of a contract) and pick the corresponding behavior from the following list." In doing so, it may perform some method instrumentation that in turn may need to declare some variables that are local to the method. To support this, each adaptive behavior description is one of the following types:

- **`call`** *`fully_scoped_method_name`* **`:`**
  `method_variable_declarations`
  `method_instrumentation`
  `region_behavior_list default_behavior`
- **`return`** *`fully_scoped_method_name`* **`:`**

Behavior Statement	Explanation
pass_through;	Passes the method call through to the default remote object.
pass by method_name;	Invokes the alternate method *method_name* on the target object.
pass to *name;*	Invokes the target method on the alternate object *name.*
pass [to target_name**] [by** method_name**] with** args	Alters the parameter mapping of the call. This can be used in conjunction with redirecting to an alternate method and/or object.
rebind rebind_spec;	Alters the remote references in the delegate.
throw *exception;*	Throws *exception* without making a remote call.
java_code { *code ...* **}**	Inserts *code* into the generated Java delegate.
cplusplus_code { *code ...* **}**	Inserts *code* into the generated C++ delegate.

Table 1 : SDL behavior statements

```
method_instrumentation
region_behavior_list default_behavior
```

The first form is used to specify the adaptive behavior in the call context (i.e., when the client calls a remote method). The second one specifies the adaptive behavior in the return context (i.e., upon returning from a remote method call). Note that method variables cannot be declared in the return context. This is because the scope of the variables declared in the call context spans the whole method, i.e., both the call and the return.

The `method_variable_declaration` provides the ability to declare variables that are local to that method. The `method_instrumentation` provides the SDL support required for QuO instrumentation. The *region_behavior_list* is a list of statements of the following form:

```
region fully_scoped_region_name :
  behavior_statement_list
```

Note that an SDL file can declare multiple contracts in the `contracts` list in its header. The *fully_scoped_region_name* is a list of nested regions in one of the contracts in that list, separated by the IDL separator ('::'). The first *region_name* in the list must always match a top-level region in one of the contracts, and each subsequent *region_name* must match a region nested within the enclosing region. Note that there is no need to explicitly specify the contract name when describing a fully scoped region name. The code generators search the contract list for regions defined matching those in an adaptive behavior description.

The *behavior_statement_list* specifies the behavior to be performed for the method and region. Table 1 lists the available SDL behavior statements.

The *default_behavior* is a statement of the form:

```
• • •
call read:
  value_seq v1,v2,v3;
  Instrumentation::PerCall::Value_seq v1;
  string ConnId;
region InstrumentationOn:
  ...
```

Figure 15: Example method variable declarations in SDL

default : *`behavior_statement_list`*

where *`behavior_statement_list`* is as defined above. This specifies the behavior used by the delegate for this method call and any regions not specified in the *`region_behavior_list`*.

Method Variable Declaration. Local variables can be defined in methods as follows:

<type_name> <variable_name1> [, *<variable_name2>* , ... *<variable_nameN>*] ;

The SDL fragment in Figure 15 describes some example method variable declarations. These declarations result in variables local to the corresponding method in the delegate class. There is no bind option for these variables.

Method Instrumentation. SDL provides two simple ways to insert instrumentation code into the generated delegate, described in this section.

In either the call or return context, one can specify verbatim code that will be used to instrument the method. This goes right before the region specific descriptions and the specified verbatim code could either be marked as a *probe_point* or as a *sig_init*. The SDL fragment in Figure 16 presents an example. In this fragment the call and return context of only one method (read) is shown and the details of the region specific behavior is omitted.

The verbatim code marked as *sig_init* is used to initialize the properties field of SignalEvent that is used as an argument for the function that performs contract evaluation.

The verbatim code marked as *probe_point* gets inserted in the method body of the delegate class right before contract evaluation takes place in the call or return context. This is used as a mechanism to instantiate and use the method variables. The code inserted this way gets executed whenever a CORBA invocation goes through a delegate. Using this kind of code instrumentation one can perform various measurements and control operation on a per call basis. See the *QuO Toolkit Version 2.1 User's and Programmer's Guide* for more details about instrumentation.

Redirecting method calls. The Pass By and Pass To features can be used to direct a method call to a different target. Pass By invokes a different method on the same target object.

As illustrated in Figure 17a, when the client calls the method `read`, the read method will be dispatched to the delegate's default remote object without any parameter mapping for all contract regions except for *AccessSuspect*. When the current contract region is *AccessSuspect*, then instead of read method, *authenticated_read* will be dispatched to the delegate's remote object. The actual parameters of the call via *authenticated_read* will be the same parameters with which read was invoked. Note that in order for this to succeed, *read* and *authenticated_read* must have conforming signatures and *authenticated_read* must be available in the interface.

```
Delegate behavior for interface InvInstrumented and
                        contracts InstrumentedServer is
  //SDL variables
  Inv altRemoteObj1 bind with name InventoryServer;
  Instrumentation inst bind with
     java_code{ new StandardInstrumentation();},
     cplusplus_code { new StandardInstrumentation();};
     //Adaptation specs for read method: call context
     call read:
       //method variables
       Method_id m;
       Probe_point java_code { m = "read";
                                    inst.probe1(m);},
                   cplusplus_code  { m = "read";
                                          inst->probe1(m);};
       sig_init java_code { inst.values(); },
                cplusplus_code { inst->values();};
       region InstrumentationOn:  ...
       region InstrumentationOff: ...
       default: ...
     //Adaptation specs for read method: return context
     return read:
       Probe_point java_code { m = "read";
                                   inst.probe2(m);},
                   cplusplus_code  { m = "read";
                                         inst->probe2(m); };
       sig_init java_code { inst.set_value(); },
                cplusplus_code { inst->set_value(); };
       region InstrumentationOn: ...
       region InstrumentationOff: ...
       default: ...
     //SDL default:
     default: ...
end delegate behavior;
```

Figure 16: Example of using SDL to instrument code

PASS TO allows us to dispatch to alternative remote objects instead of the default remote object. Figure 17b provides an example. When the client invokes *read* and the current region is *AccessSuspect*, then *read* is invoked on the alternate remote object *authenticating_server*. The parameters of this dispatched call are the same as the parameters with which *read* was invoked.

It is possible to combine PASS TO and PASS BY to dispatch a different method to an alternative remote object, as illustrated in Figure 17c. When the *read* method is invoked and the current contract region is *AccessSuspect*, the delegate will actually dispatch the *authenticated_read* method on the remote object *authenticating_server*. The actual parameters of the dispatched call will be the same as the parameters with which *read* was invoked.

Parameter Mapping. In conjunction with PASS TO and/or PASS BY it is now possible to do parameter mapping. This is specified by means of a `with` clause in the PASS TO and/or PASS BY specification:

```
pass [to <obj>] by <method> with args;
pass to <obj> [by <method>] with args;
```

The `args` can be a list of identifiers or one of the following operations on a special variable `this_method_params`:

```
call read:
  region AccessSuspect:
    pass by authenticated_read;
  default:
    pass_through;
```

(a) Example of Pass By

```
call read:
  region AccessSuspect:
    pass to authenticating_server;
```

(b) Example of Pass To

```
call read:
  region AccessSuspect:
    pass to authenticating_server
         by authenticated_read;
```

(c) Example of Pass To and By

Figure 17: Examples of redirecting method calls

```
append()
prepend()
subset()
```

Figure 18 illustrates several example variations. The special variable `this_method_params` denotes a reserved object that always refers to the actual parameters of the currently invoked method. For example, if the currently invoked method had actual parameters *x, y, z* respectively, the current value of `this_method_params` is *(x, y, z).*

If a comma separated list of identifiers are provided in the `with` clause then that list becomes the actual parameters of the dispatched call. For instance in example 1 in Figure 18, the dispatched call looks like:

```
quo_remoteObj.read(a1,a2,a3);     // Java
quo_remoteObj->read(a1,a2,a3);    // C++
```

Examples 2, 3 and 4 in Figure 18 illustrate using the functions `append()`, `prepend()` or `subset()`. Assuming that `this_method_params` is *(x, y, z)*, the dispatched call for example 2 is:

```
altRemoteObj1.bar(x, y, z, a1, a2);   // Java
altRemoteObj1->bar(x, y, z, a1, a2);  // C++
```

```
pass by read with a1,a2,a3;                                           //1
pass to altRemoteObj1 by bar with this_method_params.append(a1,a2);   //2
pass to altRemoteObj1 by bar with this_method_params.prepend(a1,a2);  //3
pass to altRemoteObj1 by bar with this_method_params.subset(0,1);     //4
```

Figure 18: Examples of parameter mapping

For example 3 it is:

```
altRemoteObj1.bar(a1, a2, x, y, z);    // Java
altRemoteObj1->bar(a1, a2, x, y, z);   // C++
```

Finally, for example 4 it is:

```
altRemoteObj1.bar(x);   // Java
altRemoteObj1->bar(x);  // C++
```

The two arguments of `subset` *(from_including, upto_excluding)* define the subset of `this_method_params`. The second argument is optional. If it is not specified it means the rest of the list starting from the first argument. The first argument cannot be less than 0, if the second argument is larger than the length of `this_method_params` it is treated as if it was not specified. As in all the alternative behavior descriptions, the method signature must match the arguments being passed to it.

Remote References. Reestablishing the remote references maintained by the delegate is possible using the *rebind* feature of SDL, which can be used in both call and return contexts. SDL supports the following uses of *rebind*:

```
rebind;
rebind x;
rebind to Interface;
rebind x to Interface;
rebind with name objName;
rebind x with name Objname;
rebind to Interface with name ObjName;
rebind x to Interface with name ObjName;
```

Rebind variations without name (i.e., those that do not have *with name ObjName* in the rebind clause) are not supported in C++. This is because the *bind* feature of Visibroker is not part of the defined CORBA standard. In ORBs that do not provide it, one needs to obtain an IOR in some other manner, such as using CORBA's naming or trading service.

When the variable name is not provided in the rebind clause (e.g., `rebind;`), it means that the delegate's default remote object will be rebound. For all cases where a name is available (i.e., *with name objName* is present in the rebind clause), the object reference is reestablished by resolving `objName` and narrowing it to the appropriate type. For all other cases an appropriate `bind()` method provided by Visibroker is used.

7.3. CSL Syntax and Semantics

Each CSL file consists of the following code sections:

- Connector Attribute Specifications
- Usage Specification of the *connectparams* Connector Attributes
- Include Statements
- Object Definitions
- Function Calls

Attribute	Sample value	Possible values	Description
target	“client”	“client”, “server”	Specifies whether the connector code and delegate code is for a client side or server side delegate.
class	“MyConn.java”	String	Name of the connector class being generated in the target language. The connector class is an implementation of the interface/abstract class specified by **interface**
interface	“ConnInf.java”	String	Name of the connector interface class being generated in the target language. The connector interface class is an interface/abstract class for the implementation class specified by **class.**
target_interface	myIDLmodule::myServerSideObj	Fully qualified IDL interface name	For server side connectors, specifies the IDL interface name of the server side delegate
connectparams	(in mod::type obj, in org::omg::CORBA::ORB myorb, in boolean gui)	Parameter list in the syntax of CORBA IDL	Specifies the signature of the connect method that is generated from the CSL code. The syntax is the same as CORBA IDL syntax.
codeloglevel	“high”	“none”, “low”, “high”	Determines the amount of debugging or logging messages are included in the generated connector code.
cppinclude	“quo.h”	String	Specifies the include files for C++ connector classes. Multiple includes are separated by spaces.
language	“java”	“java”, “c++”	The target language generated for connectors and delegates. Note: contract and transition code is always generated in Java.
package	“my.java.pkg”	String	The package in which all generated Java classes will be placed. This works in conjunction with idl2package
idl2package	“::myModule” “my.other.pkg”	module package …	Describes a mapping between modules and packages. Identifiers indicated to be in a module in the list are mapped to the corresponding package. This works like Visibroker’s idl2package feature.
idlpreproc_include	“myIDLdir/” “quo/”	String …	The include path for the IDL preprocessor. The IDL preprocessor looks in these directories for included IDL files. Multiple includes are separated by spaces.
idlpreproc_only	“yes”	“yes”, “no”	If yes, runs the IDL preprocessor only and sends the output to stdout
idlpreproc_location	“usr/local/bin”	String (pathname)	The path specifying the location of the IDL preprocessor.

Table 2: Connector attribute specifications in CSL

- Publishing Object References

Connector Attribute Specifications. Connector Attribute Specifications are used to define static properties of the connector. They can be specified in either of the following ways:

- on the command line at the time of invocation of quogen
- in the **Connector Attribute Specifications** section of the CSL file

Command line argument	Description
-l language	Specifies the target language generated for connectors and delegates. Note: contract and transition code is always generated in Java. Overrides the **language** attribute in CSL.
-p package	Specifies the package in which all generated Java classes will be placed. This works in conjunction with idl2package. Overrides the **package** attribute in CSL.
-idl2package ::module package	Describes a mapping between modules and packages. Identifiers indicated to be in a module in the list are mapped to the corresponding package. This works like Visibroker's idl2package command line argument and overrides the **idl2package** attribute in CSL.
-I includedir	The include path for the IDL preprocessor. The IDL preprocessor looks in these directories for included IDL files. This overrides the **idlpreproc_include** attribute in CSL.
-E	Runs the IDL preprocessor only and sends the output to stdout. Overrides the **idlpreproc_only** attribute in CSL.
-Yp,<path>	Specifies the location of the IDL preprocessor. Overrides the **idlpreproc_location** in CSL.
-d prefixdir	appends prefixdir to the filenames of all generated files
-O [none | errors | warnings | data | all]	sets the level of debug information being produced during code generation
-V	Prints version information
-v	Traces compilation stages
-u	Prints usage information
-w	Suppresses IDL compiler warning messages
-M	Reports all files being generated to stdout without generating the files
-C	Suppresses typechecking; continues code generation even after encountering typechecking errors

Table 3: Code generator command line arguments

Table 2 gives a summary of the connector properties that can be specified in the **Connector Attribute Specification** Section of CSL. Table 3 summarizes the command line arguments that are accepted by the code generator. In the case of conflicting specifications (e.g., 'quogen -l java...' and 'language "c++"'), the command line arguments override the corresponding CSL entries.

Usage specification of connectparams. The Usage Specification of Connectparams section specifies how each connector argument (specified by the **connectparams** part of the Connector Attribute Specification section in CSL) is used. It specifies whether each argument is to be used as an ORB, a Remote Object, the QuO Kernel, a Callback object, a System Condition object, a Value of Kernel Debug or Kernel GUI. If there is no **connectparams** attribute specification then there is no need for this section in the CSL file.

This section consists of several statements of the form:

```
use <identifier> as [ quokernel | kerneldebugval | kernelguival |
                      syscond | callback | remoteobj | orb ]
```

where <identifier> is the name of a parameter specified in *connectparams*. Figure 19 shows an example of the *connectparams* and *use* statements.

```
connectparams (in boolean kg, in long kd, in quo::ValueSC v1,
               in quo::ValueSC v2, in CORBA::ORB o,
               in quo::Callback c1, in InvInstrumented m,
               in quo:: QuoKernel qk)
...
use qk as quokernel
use kg as kernelguival
use kd as kerneldebugval
use v1 as syscond
use v2 as syscond
use o as orb
use c1 as callback
use m as remoteobj
```

Figure 19: Example of connectparams and use statements in CSL

Include statements. The **Include section** of CSL specifies the CDL, SDL, and IDL files that comprise a QuO application.

CSL Object specifications. You can specify the following five different types of objects in CSL objects statements:

- remote CORBA objects
- system condition objects
- callback objects
- contract objects
- delegate objects.

We refer to these collectively as CSL objects. CSL provides the following different ways to describe how an object is located or instantiated:

1. **Local instantiation**
   <type> <objectname> = new <implementationType> (<params>);
   Example: Counter myRemoteObj = new CounterImpl ();
2. **Reading an IOR reference from a file**
   <type> <objectname> = fileior (<filename>);
   Example: Counter myRemoteObj = fileior ("Counter.ior");
3. **Calling a factory method**
   <type> <objectname> = funcall <other_objectname>.<functionname>(<params>) ;
   Example: Counter myRemoteObj = funcall nameServer.getObject("MyCounter");
4. **Getting an appropriately narrowed reference to a CORBA object from a method call**
   <type> <objectname> = narrowfuncall <other_objname>.<fnname>(<params>) ;
   Example: Counter myRemObj = narrowfuncall someObj.getObj("MyCounter");
5. **Getting an appropriately narrowed reference to a CORBA object from a name-server**
   <type> <objectname> = resolveName (<nameserver>,<String>) ;
   Example: Counter myRemoteObj = resolveName (corbaNameServer, "MyCounter");

Remote objects are recognized by having a type defined as an interface in one of the included IDL files. *System condition objec*ts are CORBA objects for an IDL interface that extends quo::SysCond. *Callback objects* are CORBA objects for an IDL interface that extends quo::Callback. *QuO contracts* are defined using CDL (see the CDL Reference Guide). The instantiation of each contract object is defined by the instantiation of the contract class (i.e., the name of the contract class as specified in CDL) and is passed the system condition objects and callback objects that the contract uses. *Delegates* are defined by a combination of SDL descriptions (which describe delegates' behaviors), IDL descriptions (which describe delegates' signatures and types), and CDL descriptions (which describe the delegates' decision possibilities).

The connector object is a subclass of the top-level delegate class (i.e., the delegate that the client or server accesses). There might be many layers of delegates. CSL provides the following set of commands to initialize delegates:

- returndelegate <objectname> (<remote_objectname>, <contract_objectname>);
- makedelegate <objectname> (<remote_objectname>, <contract_objectname>);

The *returndelegate* command instantiates the top-level delegate. The *makedelegate* command instantiates lower-level, i.e., not top-level, delegates.

The following pseudocode demonstrates how to initialize a delegate for an IDL interface and three contracts. Assume *r* is a reference to a remote object, and *c1*, *c2*, and *c3* are instantiated contracts.

```
makedelegate delegate_bottom (r, c3);
makedelegate delegate_middle (delegate_bottom, c2);
returndelegate delegate_top (delegate_middle, c1);
```

CSL will instantiate delegate_top, delegate_middle, and delegate_bottom, initialize them, and hook them up as shown. The client will make calls to delegate_top as if it were the remote object stub. Delegate_top will pass calls through to delegate_middle as if it were the remote object stub, and delegate_middle will pass calls through to delegate_bottom as if it were the remote object. If system state, as measured by the three contracts, dictates that the call should be passed through to the remote object, then delegate_bottom will pass the call through to the actual stub for *r*.

The server side top-level delegate is an implementation for the remote interface, rather than a wrapper around the stub as on the client side, so the `<remote_objectname>` argument is omitted.

Function Calls. Once references to all CSL objects have been instantiated, function calls can be performed on them and on any parameters to the connect method. Typical uses of function calls include initialization calls (e.g., use of accessor methods on delegates to initialize variable bindings) and calls on name server objects.

The syntax of function calls is as follows:

```
funcall <objectname> . <functionname> ( <functionargs> );
```

The method <functionname> is executed on the object <objectname> with parameters <functionargs>.

7.3.1. Class Hierarchy of Code Generated from CSL

The QuO code generator generates the following files from the CSL specification:

- A file (either C++ or Java) containing the top-level interface for the connector. The filename and the class name are derived from the value specified for the **interface** variable in the CSL **Connector Attribute Specifications** section.
- A file (either C++ or Java) containing the connector class. The programmer uses the class variable in the CSL Connector Attribute Specifications section to specify the name of the class. The class inherits from the interface class and from the top-level delegate.
- Java files for the contracts and transitions specified by the CDL files included in the CSL file.
- C++ or Java code for the delegates referenced in the CSL code and specified in SDL.

8. Bibliography and References

8.1. Papers referenced in this report

[BBN98] BBN Distributed Systems Research Group, DIRM project team. *DIRM Technical Overview.* Internet URL http://www.dist-systems.bbn.com/projects/DIRM.

[Cuk98] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. Sanders, D. Bakken, M. Berman, D. Karr, and R.E. Schantz. "AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects." In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, October 1998.

[DoD85] DoD. *Trusted Computer System Evaluation Criteria.* United States Department of Defense, Washington, D.C., December 1985.

[Dyn98] Dynacorp IS. Results of the DARPA evaluation of intrusion detection systems. Internet URL http://www.dyncorp-is.com/darpa/meetings/id98dec/Files/mit-ll.pdf.

[Kic96a] G. Kiczales. "Aspect-oriented Programming." *ACM Computing Surveys*, 28A(4), December 1996.

[Kic96b] G. Kiczales. "Beyond the Black Box: Open Implementation." *IEEE Software*, January 1996.

[Kim94] G. Kim and E. Spafford. "The Design and Implementation of Tripwire: A Filesystem Integrity Checker." In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, 1994.

[Net] Netscape Corporation. *Secured Socket Layer.* Internet URL http://home.netscape.com/security/techbriefs/ssl.html.

[Sab99] C. Sabnis, M. Cukier, J. Ren, P. Rubel, W. Sanders, D. Bakken, and D. Karr. "Proteus: A Flexible Infrastructure to Implement Fault Tolerance in AQuA." In *Proceedings of the IRIP International Working Conference on Dependable Computing for Critical Applications*, January 1999.

[Sch98] D. Schmidt, D. Levine, and S. Mungee. "The Design of the TAO Realtime Object Request Broker." *Computer Communications*, 21(4), April 1998.

[Sta98] S. Staniford-Chen, B. Tung, and D. Schnackenberg. "The Common Intrusion Detection Framework." In *Proceedings of the Information Survivability Workshop*, October 1998.

[Ste99] D.F. Sterne, G.W. Tally, C.D. McDonell, D.L. Sherman, D.L. Sames, P.X. Pasturel, and E.J. Sebes. "Scalable Access Control for Distributed Object Systems." In *Proceedings of the 8th Usenix Security Symposium*, August 1999.

[Sto97] S. Stolfo, A. Prodromidis, S. Tselepis, W. Lee, D. Fan, and P. Chan. "JAM: Java Agents for Meta Learning over Distributed Databases." In *Proceedings of KDD-97 and AAI 97 Workshop on AI Methods in Fraud and Risk Management*, 1997.

8.2. Papers published on the research undertaken in this project

P.P. Pal, J.P. Loyall, R.E. Schantz, J.A. Zinky, R. Shapiro, and J. Megquier. "Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration." In *Proceedings of ISORC 2000, The Third IEEE International Symposium on Object-Oriented Real-time Distributed Computing*, March 15-17, 2000, Newport Beach, CA.

P.P. Pal, J.P. Loyall, R.E. Schantz, J.A. Zinky, and F. Webber. "Open Implementation Toolkit for Building Survivable Applications." In *Proceedings of DISCEX 2000, the DARPA Information Survivability Conference and Exposition*, January 25-27, 2000, Hilton Head Island, SC.

J.P. Loyall, P.P. Pal, R.E. Schantz, and F. Webber. "Building Adaptive and Agile Applications Using Intrusion Detection and Response." In *Proceedings of NDSS 2000, the Network and Distributed System Security Symposium*, February 2-4 2000, San Diego, CA.

R.E. Schantz, J.A. Zinky, D.A. Karr, D.E. Bakken, J. Megquier, and J.P. Loyall. "An Object-level Gateway Supporting Integrated-Property Quality of Service." In *Proceedings of ISORC '99, The 2nd IEEE International Symposium on Object-oriented Real-time Distributed Computing*, May 2-5, 1999, Palais du Grand Large 35 407 Saint-Malo, FRANCE.

R. Vanegas, J.A. Zinky, J.P. Loyall, D.A. Karr, R.E. Schantz, and D.E. Bakken. "QuO's Runtime Support for Quality of Service in Distributed Objects." In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98),* 15-18 September 1998, The Lake District, England.

J.P. Loyall, D.E. Bakken, R.E. Schantz, J.A. Zinky, D.A. Karr, R. Vanegas, and K.R. Anderson. "QoS Aspect Languages and Their Runtime Integration." *Lecture Notes in Computer Science*, Vol. 1511, Springer-Verlag. *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, 28-30 May 1998, Pittsburgh, Pennsylvania.

J.P. Loyall, R.E. Schantz, J.A. Zinky, and D.E. Bakken. "Specifying and Measuring Quality of Service in Distributed Object Systems." In *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98)*, 20-22 April 1998, Kyoto, Japan.

☆U.S. GOVERNMENT PRINTING OFFICE: 2001-610-055-10093

MISSION OF AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.